

Конспекты по практикуму ЭВМ, 1 семестр

Системы контроля версий. Виды, история развития. Решаемые задачи.....	3
Общие сведения.....	3
Основные понятия	4
Виды VCS, история развития	4
Решаемые задачи VCS	12
Основные понятия и команды. Дополнительные возможности сетевых ресурсов на примере Github.....	12
Принципы работы с git	12
Основные понятия и команды	13
Дополнительные возможности сетевых ресурсов на примере Github	19
Основные этапы построения исполняемого модуля из исходного текста программы.	
Шаги трансляции программы для языка С.....	19
Этапы для запуска программы на примере gcc c++	20
Препроцессор	20
Препроцессор C/C++.....	20
Фазы компилятора	21
Виды оптимизаций с конкретными примерами	24
Генерация кода	26
Ассемблер	26
Компоновщик	27
Загрузчик программ.....	28
Роль и принцип работы редактора связей (linker) при раздельной трансляции.....	29
Раздельная компиляция	29
Смысл кратко	29
Небольшая историческая справка	30
Пример раздельной трансляции: утилита make	31
Разница между объявлением и определением	31
Про редактор связей	32
Данные. Представление данных в памяти ЭВМ: целые числа (3 способа представления чисел со знаком). Двоично-десятичные числа. Арифметика с насыщением.....	35
Данные.....	35
Представление целых чисел в памяти ЭВМ.....	35
Числа со знаком.	36
Двоично-десятичные числа.....	43
Арифметика с насыщением.....	44
Представление данных в памяти ЭВМ: представление вещественных с плавающей точкой, специальные значения (NaN, Inf).	45
Введение.	45

Экспоненциальная запись числа.....	46
Числа с плавающей точкой. Стандарт IEEE-754.....	46
Арифметика с фиксированной точкой. Точность и ошибки в вычислениях в сравнении с float,double.....	49
Определение	49
Некоторая проблема Округления	51
Сложение и вычитание.....	51
Умножение	51
Деление	52
Преобразование коэффициента масштабирования.....	52
Преобразование с числами с плавающей запятой.....	53
Точность и ошибки в вычислениях в сравнении с float,double.....	53
Общие плюсы и минусы.....	54
Сравнение с числами с плавающей запятой	54
Применение	55
Материалы.....	56
Представление данных в памяти ЭВМ: литералы (символы и строки). Кодировки (code pages). ASCII-Z. UTF-8.....	56
Символы	56
Unicode.....	58
Строки	60
(Язык С) Типы. Представление значений в памяти ЭВМ. Адресная арифметика.....	62
Типы данных в С	62
Простые типы	62
Составные типы	63
Строки	63
Alignment	64
Адресная арифметика	65
Проблемы ручного управления памятью. Способы предотвращения, средства обнаружения. Внешняя и внутренняя фрагментации свободного пространства.....	66
Ручное управление памятью	66
API стандартной библиотеки для динамического выделения и освобождения памяти	66
С большой силой приходит большая ответственность	67
А что делать то?	68
А как обнаружить?	69
Фрагментация	70
(Язык С) Процедуры. Параметры. Глобальные и локальные переменные. Понятия области видимости, «времени жизни», размещение в памяти. Организация кадра.	
Соглашения о вызовах (способы передачи аргументов).....	71
Основные понятия	72
Подпрограмма или процедура ?	72

Организация кадра	74
Выделение памяти: стековый аллокатор, implicit free list	78
Stack Allocator	78
Самый простой аллокатор: Linear Allocator	79
Stack allocator	80
Implicit free list	82
Implicit Free Lists - что получили?	85
Выделение памяти: метод двоичных близнецов, <i>SLAB-аллокатор</i>	85
Предисловие (про segregated free lists, segregated fits)	85
Метод близнецов	87
SLAB-allocator	90
Сама суть	91
Выделение памяти блоками фиксированного размера (explicit free list)	96
Описание	96
Аллокация блока	98
Освобождение блока	98
Примеры освобождения с подходом LIFO	99
Итоги	102
Симметричное шифрование: принцип организации, преимущества и недостатки. Сети Фейстеля. Алгоритмы шифрования DES и 3DES.	102
Асимметричное шифрование: принципы организации. Алгоритм шифрования RSA.	
Другие алгоритмы асимметричного шифрования.	106
Асимметричное шифрование	106
RSA	107
Другие алгоритмы	108
Протокол Диффи-Хеллмана(Меркля). Принципиальное устройство цифровой подписи.	108
Диффи-Хеллман	108
Цифровая подпись	109
PKI	109

Системы контроля версий. Виды, история развития. Решаемые задачи.

Общие сведения

Система управления версиями (системы контроля версий, *Version Control System*) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое. Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой

программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов.

Ситуация, в которой электронный документ за время своего существования претерпевает ряд изменений, достаточно типична. При этом часто бывает важно иметь не только последнюю версию, но и несколько предыдущих. В простейшем случае можно просто хранить несколько вариантов документа, нумеруя их соответствующим образом. Такой способ неэффективен (приходится хранить несколько практически идентичных копий), требует повышенного внимания и дисциплины и часто ведёт к ошибкам, поэтому были разработаны средства для автоматизации этой работы.

Основные понятия

- Репозиторий.** Каталог, в котором хранится файловая система проекта. Для каждого проекта создаётся отдельный репозиторий. Существуют локальные и удалённые репозитории. В первом осуществляется работа над проектом на компьютере, а второй выступает в роли хранилища.
- Ветка (branch).** Дочерняя версия основного репозитория. Она входит в его состав, но не влияет на работу. После того, как разработчики закончат работу над новой функцией или исправят все баги, можно совместить дочерний и родительский репозитории.
- Коммит.** Операция позволяет зафиксировать текущее состояние проекта. После выполнения команды через консоль или использования браузерной версии Git, новая версия добавляется в репозиторий.
- Форк.** Копия репозитория, которую можно использовать для изменения исходного кода без отправки изменений в основной репозиторий. Форки часто применяют для open-source проектов, когда любой разработчик может собрать свой проект на основе готового ядра.
- Пул и пуш.** Первая операция позволяет выкачивать содержимое репозитория на компьютер, а вторая отправляет измененные файлы на сервер.
- Мастер.** Основная ветка репозитория, в которой хранится ядро проекта. В неё добавляют изменения только после тщательного тестирования.
- Кодревью.** Процесс проверки кода на соответствие техническому заданию или требованиям внутри команды. Когда один разработчик хочет добавить свой код в ядро, остальные члены команды проверяют его и если проблем нет, происходит обновление главной ветки.

Виды VCS, история развития

Обычно системы управления версиями делятся на три вида, представляющих собой три поколения в истории развития VCS.

Первое поколение: локальные системы

Системы контроля версий (VCS) первого поколения отслеживали изменения в отдельных

файлах, а редактирование поддерживалось только локально и одним пользователем за раз. Системы строились на предположении, что все пользователи будут заходить по своим учётным и записям на один и тот же общий узел Unix.

- Примеры

SCCS

SCCS(Source Code Control System) считается одной из первых успешных систем управления версиями. Она была разработана в 1972 году Марком Рочкинлом из Bell Labs. Система написана на C и создана для отслеживания версий исходного файла. Кроме того, она значительно облегчила поиск источников ошибок в программе. Базовая архитектура и синтаксис SCCS позволяют понять корни современных инструментов VCS.

Команды, предоставляемые SCCS:

1. Внесение (check-in) файлов для отслеживания истории в SCCS.
2. Извлечение (check-out) конкретных версий файлов для ревью или компиляции.
3. Извлечение конкретных версий для редактирования.
4. Внесение новых версий файлов вместе с комментариями, объясняющими изменения.
5. Отмена изменений, внесённых в извлечённый файл.
6. Основные ветвления и слияния изменений.
7. Журнал изменений файла.

Поскольку содержимое исходного файла теперь хранится в файле истории, его можно извлечь в рабочий каталог для просмотра, компиляции или редактирования. В файл истории можно внести изменения, такие как добавления строк, изменения и удаления, что увеличивает его номер версии.

Последующие добавления файла хранят только изменения, а не всё его содержимое. Это уменьшает размер файла истории. Поскольку файлы истории SCCS не используют сжатие, они обычно имеют больший размер, чем фактический файл, в котором отслеживаются изменения. SCCS использует метод под названием чередующиеся дельты (interleaved deltas), который гарантирует постоянное время извлечения независимо от давности извлечённой версии, то есть более старые версии извлекаются с той же скоростью, что и новые. Важно отметить, что все файлы отслеживаются и регистрируются отдельно. Невозможно проверить изменения в нескольких файлах в виде одного атомарного блока, как коммиты в Git. SCCS поддерживает ветви, которые хранят последовательности изменений в определённом файле.

Когда файл извлекается для редактирования в SCCS, на него ставится блокировка, так что его никто больше не может редактировать. Это предотвращает перезапись изменений другими пользователями, но также ограничивает разработку, потому что в каждый момент времени только один пользователь может работать с данным файлом.

RCS

RCS (Revision Control System) написана в 1982 году Уолтером Тихи на языке С в качестве альтернативы системе SCCS, которая в то время не была опенсорсной. У RCS много общего с SCCS.

RCS позволяет работать только с отдельными файлами, создавая для каждого историю изменений. Для текстовых файлов сохраняются не все версии файла, а только последняя версия и все изменения, внесенные в нее. RCS также может отслеживать изменения в бинарных файлах, но при этом каждое изменение хранится в виде отдельной версии файла.

Когда изменения в файл вносит один из пользователей, для всех остальных этот файл остается заблокированным. Они не могут запросить его из репозитория для редактирования, пока первый пользователь не закончит работу и не зафиксирует изменения.

Преимущества RCS:

8. RCS - проста в использовании и хорошо подходит для ознакомления с принципами работы систем контроля версий.
9. Хорошо подходит для резервного копирования отдельных файлов, не требующих частого изменения группой пользователей.
10. Широко распространена и предустановлена в большинстве свободно распространяемых операционных системах.

Недостатки RCS:

11. Отслеживает изменения только отдельных файлов, что не позволяет использовать ее для управления версиями больших проектов.
12. Не позволяет одновременно вносить изменения в один и тот же файл несколькими пользователями.
13. Низкая функциональность, по сравнению с современными системами контроля версий.

Для хранения изменений RCS использует схему обратных дельт (reverse-delta). При добавлении файла полный снимок его содержимого сохраняется в файле истории. Когда файл изменяется и возвращается снова, вычисляется дельта на основе существующего содержимого файла истории. Старый снимок отбрасывается, а новый сохраняется вместе с дельтой, чтобы вернуться в старое состояние. Это называется обратной дельтой, так как для извлечения более старой версии RCS берёт последнюю версию и последовательно применяет дельты до тех пор, пока не достигнет нужной версии. Этот метод позволяет очень быстро извлекать текущие версии, так как всегда доступен полный снимок текущей ревизии. Однако чем старше версия, тем больше времени занимает проверка, потому что нужно проверить всё большее количество дельт.

Второе поколение: централизованные системы

В VCS второго поколения появилась поддержка сети, что привело к централизованным хранилищам с «официальными» версиями проектов. Это был значительный прогресс, поскольку несколько пользователей могли одновременно работать с кодом, делая коммиты в один и тот же центральный репозиторий. Однако для коммитов требовался доступ к сети.

- Примеры

CVS

CVS(Concurrent Versions System) создана Диком Груном в 1986 году с целью добавить в систему управления версиями поддержку сети. Она также написана на С и знаменует собой рождение второго поколения инструментов VCS, благодаря которым географически рассредоточенные команды разработчиков получили возможность работать над проектами вместе.

CVS — это фронтенд для RCS, в нём появился новый набор команд для взаимодействия с файлами в проекте, но под капотом используется тот же формат файла истории RCS и команды RCS. Впервые CVS позволил нескольким разработчикам одновременно работать с одними и теми же файлами.

Работа CVS организована следующим образом. Последняя версия и все сделанные изменения хранятся в репозитории сервера. Клиенты, подключаясь к серверу, проверяют отличия локальной версии от последней версии, сохраненной в репозитории, и, если есть отличия, загружают их в свой локальный проект. При необходимости решают конфликты и вносят требуемые изменения в разрабатываемый продукт. После этого все изменения загружаются в репозиторий сервера. CVS, при необходимости, позволяет откатываться на нужную версию разрабатываемого проекта и вести управление несколькими проектами одновременно.

Достоинства CVS:

1. Несколько клиентов могут одновременно работать над одним и тем же проектом.
2. Позволяет управлять не одним файлом, а целыми проектами.
3. Обладает огромным количеством удобных графических интерфейсов, способных удовлетворить практически любой, даже самый требовательный вкус.
4. Широко распространена и поставляется по умолчанию с большинством операционных систем Linux.
5. При загрузке тестовых файлов из репозитория передаются только изменения, а не весь файл целиком.

Недостатки CVS:

6. При перемещении или переименовании файла или директории теряются все, привязанные к этому файлу или директории, изменения.
7. Сложности при ведении нескольких параллельных веток одного и того же проекта.
8. Ограниченная поддержка шрифтов.

9. Для каждого изменения бинарного файла сохраняется вся версия файла, а не только внесенное изменение.
10. С клиента на сервер измененный файл всегда передается полностью.
11. Ресурсоемкие операции, так как требуют частого обращения к репозиторию, и сохраняемые копии имеют некоторую избыточность.

SVN

SVN(Subversion) создана в 2000 году компанией Collabnet Inc., а в настоящее время поддерживается Apache Software Foundation. Система написана на С и разработана как более надёжное централизованное решение, чем CVS.

Как и CVS, Subversion использует модель централизованного репозитория. Удалённым пользователям требуется сетевое подключение для коммитов в центральный репозиторий.

Subversion представила функциональность атомарных коммитов с гарантией, что коммит либо полностью успешен, либо полностью отменяется в случае проблемы. В CVS при неполадке посреди коммита (например, из-за сбоя сети) репозиторий мог остаться в повреждённом и несогласованном состоянии. Кроме того, коммит или версия в Subversion может включать в себя несколько файлов и директорий. Это важно, потому что позволяет отслеживать наборы связанных изменений вместе как сгруппированный блок, а не отдельно для каждого файла, как в системах прошлого.

Достоинства SVN:

12. Система команд, схожая с CVS.
13. Поддерживается большинство возможностей CVS.
14. Разнообразные графические интерфейсы и удобная работа из консоли.
15. Отслеживается история изменения файлов и каталогов даже после их переименования и перемещения.
16. Высокая эффективность работы, как с текстовыми, так и с бинарными файлами.
17. Встроенная поддержка во многие интегрированные средства разработки, такие как KDevelop, Zend Studio и многие другие.
18. Возможность создания зеркальных копий репозитория.
19. Два типа репозитория – база данных или набор обычных файлов.
20. Возможность доступа к репозиторию через Apache с использованием протокола WebDAV.
21. Наличие удобного механизма создания меток и ветвей проектов.
22. Можно с каждым файлом и директорией связать определенный набор свойств, облегчающий взаимодействие с системой контроля версии.
23. Широкое распространение позволяет быстро решить большинство возникающих проблем, обратившись к данным, накопленным Интернет-сообществом.

Недостатки SVN:

24. Полная копия репозитория хранится на локальном компьютере в скрытых файлах, что требует достаточно большого объема памяти.
25. Существуют проблемы с переименованием файлов, если переименованный локально файл одним клиентом был в это же время изменен другим клиентом и загружен в репозиторий.
26. Слабо поддерживаются операции слияния веток проекта.
27. Сложности с полным удалением информации о файлах, попавших в репозиторий, так как в нем всегда остается информация о предыдущих изменениях файла, и непредусмотрено никаких штатных средств для полного удаления данных о файле из репозитория.

Третье поколение: распределенные системы

Третье поколение состоит из распределённых VCS, где все копии репозитория считаются равными, нет центрального репозитория. Это открывает путь для коммитов, ветвей и слияний, которые создаются локально без доступа к сети и перемещаются в другие репозитории по мере необходимости.

- Примеры

Git

С февраля 2002 года для разработки ядра Linux'а большинством программистов стала использоваться система контроля версий BitKeeper. Довольно долгое время с ней не возникало проблем, но в 2005 году Лари МакВоем (разработчик BitKeeper'a) отозвал бесплатную версию программы.

Разрабатывать проект масштаба Linux без мощной и надежной системы контроля версий – невозможно. Одним из кандидатов и наиболее подходящим проектом оказалась система контроля версий Monotone, но Торвальдса Линуса не устроила ее скорость работы. Так как особенности организации Monotone не позволяли значительно увеличить скорость обработки данных, то 3 апреля 2005 года Линус приступил к разработке собственной системы контроля версий – Git.

Практически одновременно с Линусом, к разработке новой системы контроля версий приступил и Мэтт Макал. Свой проект Мэтт назвал Mercurial.

Git – это гибкая, распределенная (без единого сервера) система контроля версий, дающая массу возможностей не только разработчикам программных продуктов, но и писателям для изменения, дополнения и отслеживания изменения «рукописей» и сюжетных линий, и учителям для корректировки и развития курса лекций, и администраторам для ведения документации, и для многих других направлений, требующих управления историей изменений.

У каждого разработчика, использующего Git, есть свой локальный репозиторий, позволяющий локально управлять версиями. Затем, сохраненными в локальный

репозиторий данными, можно обмениваться с другими пользователями. Часто при работе с Git создают центральный репозиторий, с которым остальные разработчики синхронизируются. В этом случае все участники проекта ведут свои локальные разработки и беспрепятственно скачивают обновления из центрального репозитория. Когда необходимые работы отдельными участниками проекта выполнены и отложены, они, после удостоверения владельцем центрального репозитория в корректности и актуальности проделанной работы, загружают свои изменения в центральный репозиторий.

Наличие локальных репозиториев также значительно повышает надежность хранения данных, так как, если один из репозиториев выйдет из строя, данные могут быть легко восстановлены из других репозиториев.

Работа над версиями проекта в Git может вестись в нескольких ветках, которые затем могут с легкостью полностью или частично объединяться, уничтожаться, откатываться и разрастаться во все новые и новые ветки проекта.

Достоинства Git:

1. Надежная система сравнения ревизий и проверки корректности данных, основанные на алгоритме хеширования SHA1 (Secure Hash Algorithm 1).
2. Гибкая система ветвления проектов и слияния веток между собой.
3. Наличие локального репозитория, содержащего полную информацию обо всех изменениях, позволяет вести полноценный локальный контроль версий и заливать в главный репозиторий только полностью прошедшие проверку изменения.
4. Высокая производительность и скорость работы.
5. Удобный и интуитивно понятный набор команд.
6. Множество графических оболочек, позволяющих быстро и качественно вести работы с Git'ом.
7. Возможность делать контрольные точки, в которых данные сохраняются без дельта компрессии, а полностью. Это позволяет уменьшить скорость восстановления данных, так как за основу берется ближайшая контрольная точка, и восстановление идет от нее. Если бы контрольные точки отсутствовали, то восстановление больших проектов могло бы занимать часы.
8. Широкая распространенность, легкая доступность и качественная документация.
9. Гибкость системы позволяет удобно ее настраивать и даже создавать специализированные контролиры системы или пользовательские интерфейсы на базе git.
10. Универсальный сетевой доступ с использованием протоколов http, ftp, rsync, ssh и др.

Недостатки Git:

11. Unix – ориентированность. На данный момент отсутствует зрелая реализация Git, совместимая с другими операционными системами.
12. Возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по

содержанию ревизий.

13. Не отслеживается изменение отдельных файлов, а только всего проекта целиком, что может быть неудобно при работе с большими проектами, содержащими множество несвязных файлов.
14. При начальном (первом) создании репозитория и синхронизации его с другими разработчиками, потребуется достаточно длительное время для скачивания данных, особенно, если проект большой, так как требуется скопировать на локальный компьютер весь репозиторий.

Mercurial

Mercurial создан в 2005 году Мэттом Макколлом и написан на Python(только отдельные участки программы, требующие наибольшего быстродействия, написаны на языке Си). Он тоже разработан для хостинга кодовой базы Linux, но для этой задачи в итоге выбрали Git. Это вторая по популярности система управления версиями, хотя она используется гораздо реже.

Mercurial использует многие из тех же технологий, что и Git. Так же, как и в git'е, поддерживается возможность создания веток проекта с последующим их слиянием.

Для взаимодействия между клиентами используются протоколы HTTP, HTTPS или SSH.

Набор команд - простой и интуитивно понятный, во многом схожий с командами subversion. Так же имеется ряд графических оболочек и доступ к репозиторию через веб-интерфейс. Немаловажным является и наличие утилит, позволяющих импортировать репозитории многих других систем контроля версий.

Рассмотрим основные достоинства и недостатки Mercurial.

Достоинства:

15. Быстрая обработка данных.
16. Кроссплатформенная поддержка.
17. Возможность работы с несколькими ветками проекта.
18. Простота в обращение.
19. Возможность конвертирования репозиториев других систем поддержки версий, таких как CVS, Subversion, Git, Darcs, GNU Arch, Bazaar и др.

Недостатки:

20. Возможные (но чрезвычайно низкие) совпадения хеш - кода отличных по содержанию ревизий.
21. Ориентирован на работу в консоли.

Решаемые задачи VCS

1. **Защищает исходный код от потери.** Данные хранятся на удалённом сервере, даже если разработчики удалят файлы с локального компьютера, они останутся в репозитории.
2. **Обеспечивает командную работу.** Программисту не надо использовать инструменты для командной работы и платить за них. Каждый может работать на своём компьютере и обновлять файлы по мере необходимости.
3. **Помогает отменить изменения.** В любой момент можно вернуться к контрольной точке, сравнить исходный код с текущим и обновить главную ветку после реview.
4. **Распределённая работа.** Необязательно работать с проектом «наживую». Плагин может функционировать на сайте, а программисты будут спокойно создавать новую версию.

Основные понятия и команды.

Дополнительные возможности сетевых ресурсов на примере Github.

Git — распределённая система контроля версий, позволяющая сохранять изменения, внесённые в файлы, которые хранятся в репозитории. Сами изменения сохраняются в виде снимков, называемых коммитами. Они могут размещаться на разных серверах, поэтому вы всегда восстановите код в случае сбоя, а также без проблем откатитесь до любого предыдущего состояния. Кроме того, значительно облегчается взаимодействие с другими разработчиками: несколько человек могут работать над одним репозиторием одновременно, сохраняя свои изменения.

Принципы работы с git

Настройка git (.gitconfig)

Для начала, нужно настроить конфигурационный файл. Как минимум, указать name и email, ведь они используются для идентификации(каждый коммит в Git содержит эту информацию).

```
git config --global user.name "Yakov"  
git config --global user.email NGBTR@gmail.com
```

Кому интересно узнать больше про git config (узнать, что такое --global из примера): [тык](#)

Работа с проектом

Простая схема работы с git'ом:

- Подгружаем изменения, которые были сделаны командой (делаем pull)
- Делаем изменения в файлах проекта

- Фиксируем изменения (делаем commit)
- Отправляем изменения в свет (делаем push)

Основные понятия и команды

Определения

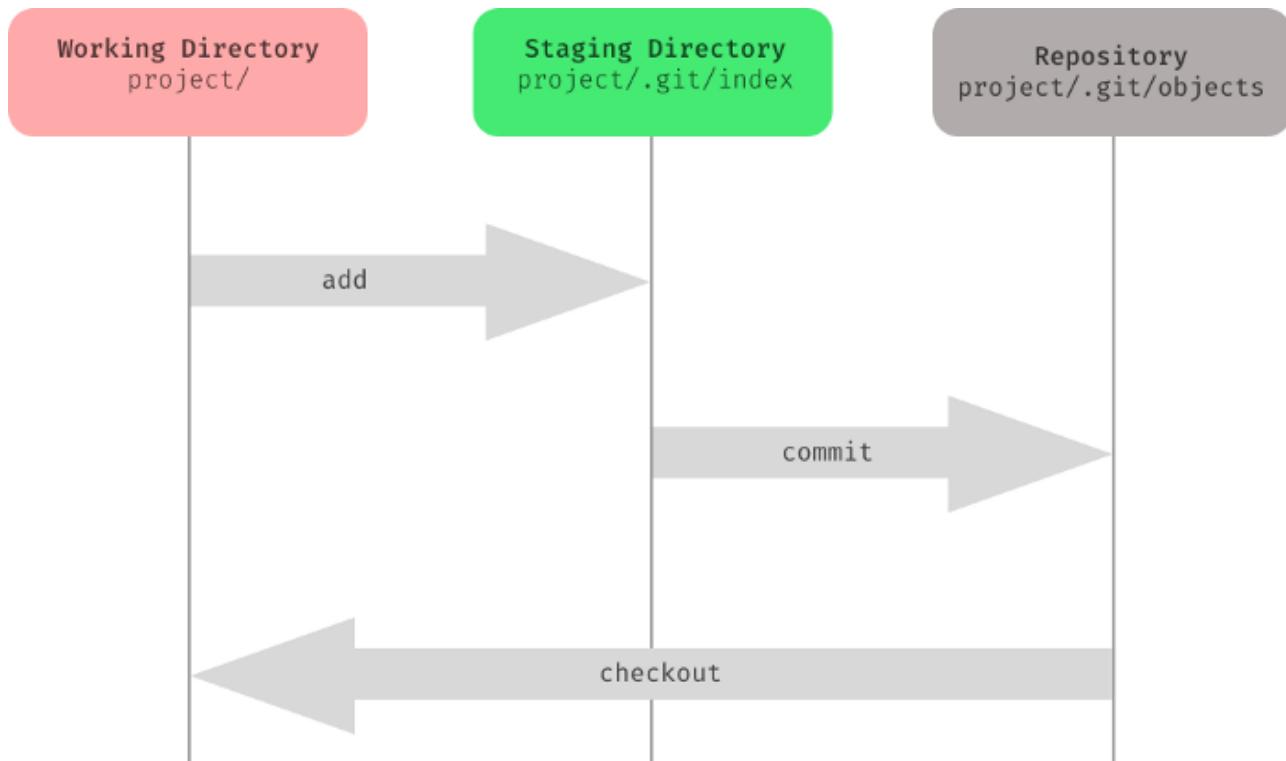
Рабочая директория (working directory) - директория на локальном компьютере, в которой находится проект. Это место, где вы можете создавать, изменять и удалять файлы в процессе разработки.

Index file (aka stage or cache file) - двоичный файл, содержащий список всех файлов, которые будут включены в следующий коммит.

- подробнее

Это промежуточный слой между рабочей директорией и локальной базой данных Git, который позволяет пользователю выбирать, какие изменения должны быть включены в следующий коммит.

После редактирования файлов в рабочей директории, пользователь может использовать команду `git add` для добавления этих изменений в index file. Когда пользователь готов закоммитить изменения в локальный репозиторий, он выполняет команду `git commit`, которая создает коммит на основе списка изменений в index file.



source: <https://konrad126.medium.com/understanding-git-index-4821a0765cf>

Index file содержит информацию о каждом измененном файле, включая его имя, время последнего изменения и состояние (изменен, добавлен или удален). Кроме того, индекс используется для отслеживания конфликтов между файлами в рабочей директории и

репозитории, что позволяет пользователям решать проблемы слияния веток и другие подобные конфликты.

[Статья о том, как устроен index file](#)

Сразу стоит сказать про состояния файлов:

- **Неотслеживаемый (untracked)** - находится в рабочей директории, но его нет ни в HEAD, ни в области подготовленных файлов. Можно сказать, что Git о нём не знает.
- **Изменён (modified)** - в рабочей директории находится его более новая версия по сравнению с той, которая хранится в HEAD либо в области подготовленных файлов (**то есть файл изменен, но его нет в index file**).
- **Подготовлен (staged)** - в области подготовленных файлов и в рабочей директории есть более новая версия, если сравнивать с хранящейся в HEAD, но файл уже готов к коммиту (**файл изменен и присутствует в index file**).
- **Без изменений (unmodified)** - во всех разделах содержится одна версия файла, то есть в последнем коммите находится актуальная версия.
 - Еще немного про **index file**

Почему некоторые измененные файлы есть в index file, а некоторые нет?

Все потому что файл добавляется в index file после того, как **мы** его туда добавим (например, с помощью команды **git add**).

[Подробнее про добавление измененных файлов в index file.](#)

Так же там написано про то, как можно проиндексировать файл другим способом(не через **git add**)

Локальный репозиторий (local repository) - это база данных Git на локальном компьютере, которая содержит все версии и историю изменений файлов проекта, а также все созданные ветки и отслеживаемые изменения. Локальный репозиторий позволяет работать над проектом, выполнять коммиты и просмотр истории изменений без подключения к удаленному репозиторию.

- подробнее

хранится в директории .git в вашем проекте.

Сегодня, 18:58	--	Папка
refs	Сегодня, 17:40	-- Папка
packed-refs	9 мая 2023 г., 02:09	182 Б Документ
ORIG_HEAD	1 мая 2023 г., 14:47	41 Б Документ
objects	Сегодня, 17:41	-- Папка
logs	Сегодня, 17:39	-- Папка
info	26 марта 2023 г., 17:16	-- Папка
index	Сегодня, 18:58	10 КБ Документ
hooks	26 марта 2023 г., 17:16	-- Папка
HEAD	9 мая 2023 г., 02:09	21 Б Документ
FETCH_HEAD	9 мая 2023 г., 02:09	220 Б Документ
description	26 марта 2023 г., 17:16	73 Б Документ
config	9 мая 2023 г., 02:09	309 Б Документ
COMMIT_EDITMSG	4 мая 2023 г., 15:38	52 Б Документ

Прошу заметить, что тут есть config! Конфигурации в нем будут перекрывать конфигурации из `~/.gitconfig`

[Дополнительно про содержание директории .git](#)

Удаленный репозиторий (remote repository) - это база данных Git, расположенная на удаленном сервере, к которому можно обращаться через сеть. Это место, где вы можете хранить код, который разрабатывается командой разработчиков, а также синхронизировать изменения между локальными копиями репозитория разных разработчиков.

Ветки (branches) - это механизм, который позволяет разработчикам работать над несколькими наборами изменений на основе общего исходного кода. Вместо того, чтобы работать над одной и той же версией кода и решать возникающие конфликты, разработчики могут создавать ветки, на которых будут работать, и в конечном итоге объединять свои изменения. Текущую локальную ветку определяет указатель **HEAD**, показывающий на последний коммит в ветке.

Команды

Обычно, команды Git имеют следующий вид:

```
git <команда> <опции> <аргументы>
```

Каждую(или почти каждую) из следующих команд вы когда-то использовали, поэтому без лишних пояснений. Если надо, то про каждую команду можно дополнительную прочитать по приложенной ссылке.

1. [git init](#) - создание нового репозитория
2. [git clone](#) - копирование репозитория на локальную машину
3. [git add](#) - добавление изменений в index file

Здесь же рекомендую прочитать про [.gitignore](#) тем, кто забыл, что это такое

4. [git commit](#) - сохранение изменений в локальной базе данных Git

5. [git push](#) - отправка изменений на удаленный репозиторий
6. [git fetch](#) - получение изменений с удаленного репозитория

Загружает изменения из удаленного репозитория, но не изменяет ваш локальный репозиторий

7. [git pull](#) - получение изменений с удаленного репозитория

Загружает **изменения** из удаленного репозитория и сразу же выполняет git merge с вашим локальным репозиторием.

8. [git branch](#) - для работы с ветками (добавить, удалить, узнать список имеющихся веток)
9. [git checkout](#) - переключение между созданными ветками и коммитами

HEAD перемещается на указатель другой ветки или коммита.

Когда мы делаем checkout не на ветку, а на коммит, то возникает такое понятие как **Detached HEADs**. Если интересно, можете почитать.

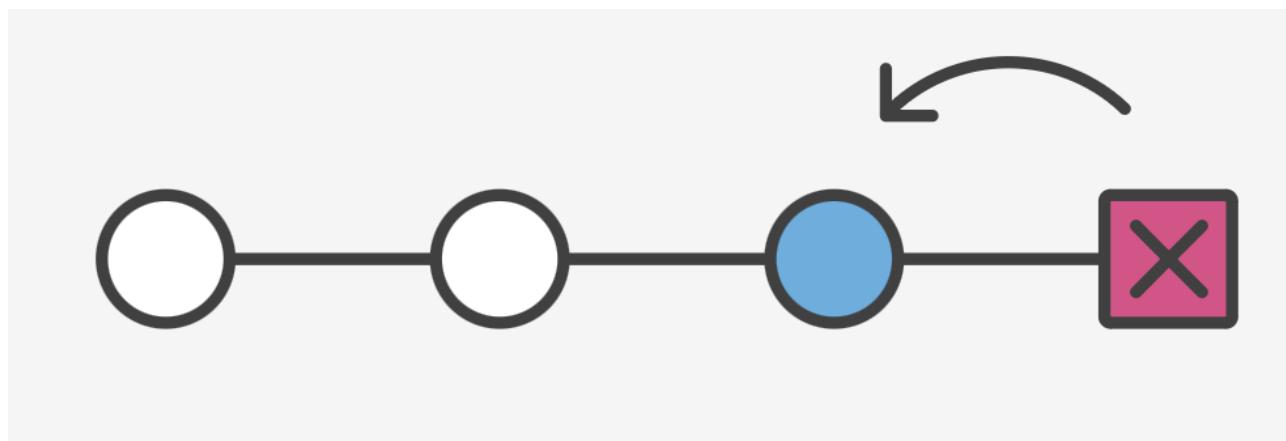
10. [git merge](#) - объединение изменений из одной ветки в другую

[Про решение конфликтов, используя консоль](#)

11. [git status](#) - проверка состояния файлов в рабочей директории
12. [git log](#) - просмотр истории коммитов в репозитории
13. [git diff](#) - просмотр изменений между двумя коммитами, ветками или файлами

На сайте по ссылке рассказывается о каждой строчке в консоли, которые выводятся при выполнении команды, о том, как делать diff между коммитами, ветками, файлами.

14. [git stash](#) - временное сохранение изменений в отдельной области, чтобы переключиться на другую ветку без коммита текущих изменений
15. [git reset](#) - отмена изменений, вернувшись к конкретному коммиту



- У **git reset** есть интересные опции —hard, —mixed, —soft.

Смотрите картинку ниже

—hard: **откатимся** к более старому коммиту, у нас **очистится** index file и **удалятся** незакоммиченные изменения (Если на примере, то пусть HEAD указывает на D. Мы написали еще пару строчек кода, которые не закоммичены, но добавлены в index file. Теперь мы делаем:

```
git reset --hard B
```

После выполнения команды, удаляются коммиты **C** и **D**. **HEAD** будет указывать на **B**. Мы получим то состояние проекта, которое было на момент коммита **B**, потеряем все, что написано в коммитах **C**, **D** и после **D**)

—mixed: **откатимся** к более старому коммиту, у нас **очистится** index file, но незакоммиченные изменения **останутся**. (Если на примере, то пусть HEAD указывает на D. Мы написали еще пару строчек кода, которые не закоммичены, но добавлены в index file. Теперь мы делаем:

```
git reset --mixed B
```

После выполнения команды, удаляются коммиты **C** и **D**. **HEAD** будет указывать на **B**. При этом если мы посмотрим в программу, у нас там **ничего** после выполнения команды не поменяется. Если пропишем git status, то увидим, что все файлы измененные в **C**, **D** и после **D** будут в состоянии **modified** или **untracked**, то есть index file почистится)

—soft: **откатимся** к более старому коммиту, index file и незакоммиченные изменения **остаются**. (Если на примере, то пусть HEAD указывает на D. Мы написали еще пару строчек кода, которые не закоммичены, но добавлены в index file. Теперь мы делаем:

```
git reset --soft B
```

После выполнения команды, удаляются коммиты **C** и **D**. **HEAD** будет указывать на **B**. При этом если мы посмотрим в программу, у нас там **ничего** после выполнения команды не поменяется. Если пропишем git status, то увидим, что все файлы измененные в **C**, **D** и после **D** будут в состоянии **staged**)



16. [git revert](#) - позволяет отменить изменения какого-то старого коммита.

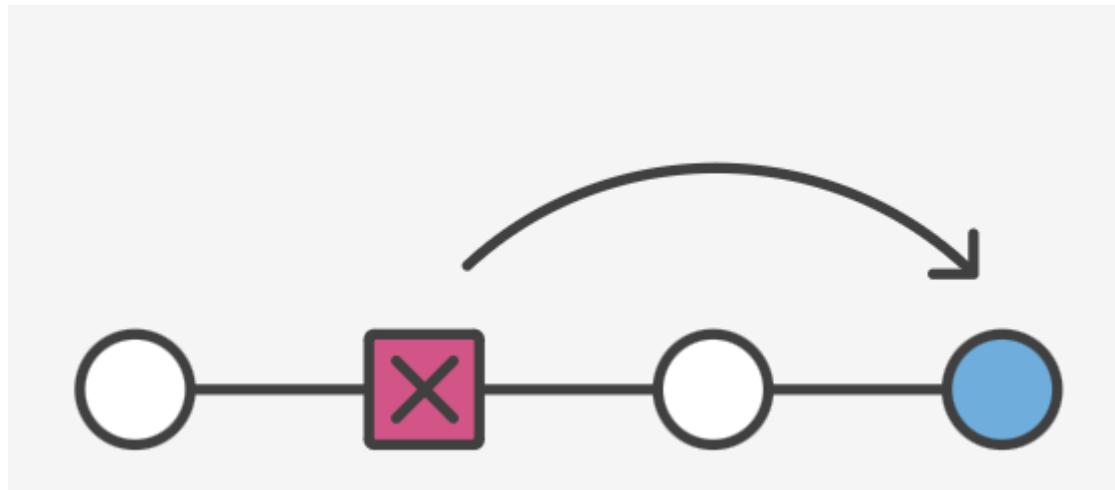
Представим, что мы находимся в ветке develop. Мы сделали в ней 3 коммита и потом

заметили багу во втором коммите. С помощью команды:

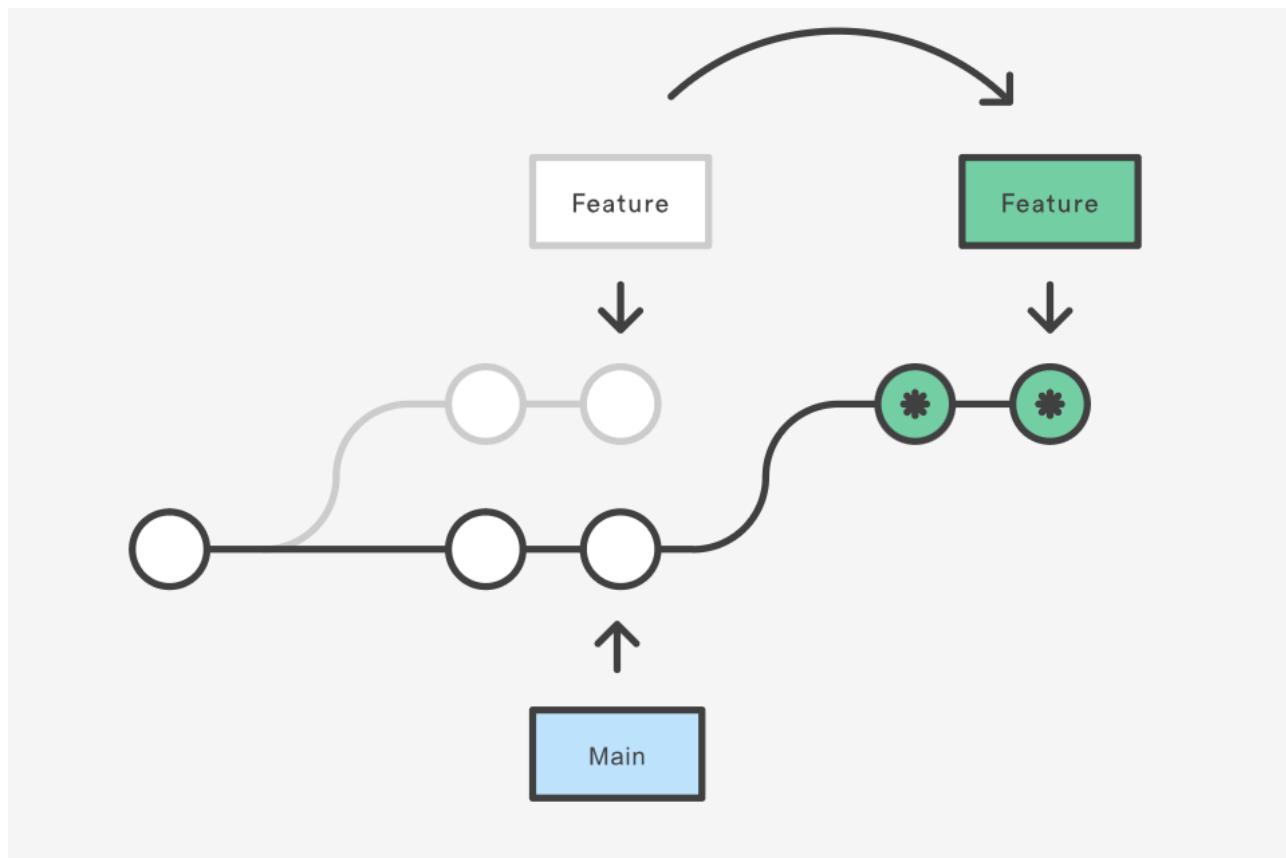
```
git revert HEAD^
```

мы можем отменить изменения из второго коммита, не отменяя изменения из третьего коммита.

Создастся новый коммит (отмечен синим кружочком на картинке), в котором отменены изменения из второго коммита.



17. [git rebase](#) - это процесс перемещения последовательности коммитов.



Дополнительные возможности сетевых ресурсов на примере Github

GitHub – один из сервисов, основанных на git, предоставляющий некоторые дополнительные возможности.

1. **Fork.** Создание ветвления от проекта без прав изменения (создание собственной копии чужого проекта на GitHub).
2. **Pull Request.** Запрос на принятие изменений (открытие обсуждения на слияние из другой ветки или ветвления).
3. **Issues.** Инструмент для уведомления об ошибках, запросах на улучшения или других проблемах, связанных с проектом. Каждую проблему можно отслеживать и комментировать, что помогает упростить процесс устранения проблем.
4. **Wiki.** GitHub предоставляет возможность создавать вики-страницы для проекта. Это может быть использовано для документирования кода, создания более подробного описания функциональности и т.д.
5. **Код-рецензирование.** Разработчики могут использовать GitHub для проведения кодревью, то есть общего анализа кода с целью идентификации проблем и нахождения путей для его улучшения.
6. **Инструменты для сотрудничества.** GitHub предоставляет инструменты для работы с командой, такие как chat-каналы, [проекты](#), уведомления и т.д., что способствует эффективному сотрудничеству.
7. **GitHub Actions.** Это встроенная в GitHub функциональность для автоматизации вашего командного процесса. Это означает, что вы можете настроить действия, которые выполняются при срабатывании определенных событий, например, когда происходят коммиты в ваш репозиторий или когда вы создаете и распаковываете версию проекта.
GitHub Actions представляет собой набор инструментов для настройки непрерывной интеграции (CI) и непрерывной развертки (CD).
8. **Статистика.** Очень много различной статистики по многим аспектам проекта.
9. **Релизы проекта.** Они есть, можете сами посмотреть!

Основные этапы построения исполняемого модуля из исходного текста программы. Шаги трансляции программы для языка С.

Этапы для запуска программы на примере gcc c++

Состав gcc c++:

- **Препроцессор (cpp)** — это макропроцессор, который преобразовывает вашу программу для дальнейшего компилирования. Обрабатывает директивы и делает макроподстановку, остальное просто копирует, умеет считать выражения.
- **Компилятор (cc1plus)**
- **Ассемблер (gas)** — специфический транслятор языка мнемоник.
- **Компоновщик (линкер, ld)** связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Объединяет модули в исполняемый файл (или в динамическую библиотеку).
- Загрузка

Препроцессор

Препроцессор — это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора). О данных на выходе препроцессора говорят, что они находятся в **препроцессированной** форме, пригодной для обработки последующими программами (компилятор). Результат и вид обработки зависят от вида препроцессора; так, некоторые препроцессоры могут только выполнить простую текстовую подстановку, другие способны по возможностям сравняться с языками программирования. Наиболее частый случай использования препроцессора — обработка исходного кода перед передачей его на следующий шаг компиляции.

Препроцессор C/C++

Препроцессор C/C++ - программа, подготавливающая код программы на языке C/C++ к компиляции.

Основные функции препроцессора

- замена соответствующих **диграфов** и **триграфов** на эквивалентные символы «#» и «\»;
 - **Диграф** — последовательность из двух или более символов, интерпретируемая компилятором как один или более символов.
 - **Триграф** — последовательность из трёх символов, первые два из которых — вопросительные знаки («??»), а третий указывает на значение триграфа
- удаление экранированных **символов перевода строки**;
- замена строчных и блочных **комментариев** пустыми строками (с удалением окружающих пробелов и символов табуляции);
- вставка (включение) содержимого произвольного файла (**#include**);
- **макроподстановки** (**#define**);

- условная компиляция (`#if`, `#ifdef`, `#elif`, `#else`, `#endif`)

Условная компиляция позволяет выбрать код для компиляции в зависимости от: **модели процессора** (платформы); разрядности адресов; **размерности типов**; наличия/отсутствия поддержки расширений языка; **наличия/отсутствия библиотек и/или функций**; особенностей поведения конкретных функций; ** и другого.

- вывод сообщений (`#warning`, `#error`).

Этапы работы препроцессора

- лексический анализ** кода C/C++ ([синтаксический анализ](#) не выполняется);
- обработка [директив](#) (`#define` и `#include`);
- выполнение подстановок:
 - диграфов и триграфов;
 - комментариев;
 - директив;
 - лексем, заданных директивами.

Язык препроцессора C/C++ не является [полным по Тьюрингу](#), следовательно, с помощью директив невозможно заставить препроцессор зависнуть. См. [рекурсивная функция \(теория вычислимости\)](#).

Фазы компилятора

Лексический анализ

Лексический анализ — процесс аналитического разбора входной последовательности символов на распознанные группы — **лексемы** — с целью получения на выходе идентифицированных последовательностей, называемых **токенами** (подобно группировке букв в словах).

На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.

Как правило, лексический анализ производится с точки зрения определённого [формального языка](#) или набора языков. Язык, а точнее, его [грамматика](#), задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть идентифицирована как

токен языка, обычно рассматривается как специальный токен-ошибка.

Синтаксический (грамматический) анализ

Синтаксический анализ — процесс сопоставления линейной последовательности лексем (слов, токенов) **формального языка** с его **формальной грамматикой**. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с **лексическим анализом**.

Формальный язык — множество конечных слов (строк, цепочек) над конечным алфавитом.

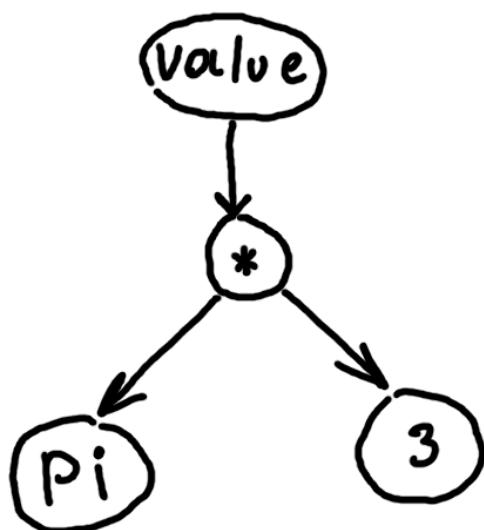
Алфавит формального языка — множество атомарных (неделимых) символов какого-либо формального языка (иногда их называют буквами по аналогии с алфавитами естественных языков или символами). Из символов алфавита формального языка строятся слова, а заданием формальной грамматики — допустимые выражения языка.

Формальная грамматика — способ описания формального языка, то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита. Различают *порождающие* и *распознающие* (или *аналитические*) грамматики — первые задают правила, с помощью которых можно построить любое слово языка, а вторые позволяют по данному слову определить, входит ли оно в язык или нет.

Дерево разбора — упорядоченное корневое дерево, представляющее синтаксическую структуру строки в соответствии с некоторой **контекстно-свободной грамматикой**.

Пример кода в виде дерева разбора:

```
value = pi * 3
```



Почитать: Грамматика зависимостей, Грамматика составляющих, Иерархия Хомского, Разные анализаторы, Форма Бэкуса — Наура

Семантический анализ

Семантический анализ — этап в последовательности действий алгоритма автоматического понимания текстов, заключающийся в выделении семантических отношений, формировании семантического представления текстов.

Основной работой семантического анализа является *проверка типов* (type checking). Например, большинство языков программирования используют целые числа для индексов массива. Компилятор должен вывести ошибку, если используется числа с плавающей точкой в качестве индекса.

Семантический анализ занимается приведением типов (type conversions/type coercions). Мы знаем, что бинарные операции должны проводиться парами значений одних типов, поэтому рассмотрим следующий пример:

```
value = pi * 3
```

pi - переменная типа float, она умножается на целочисленное 3. Семантический анализатор замечает это и конвертирует целое число в число с плавающей точкой. (может поставить условно функцию **inttofloat**, которая и выполняет конвертацию)

Генерация кода промежуточного представления

Дерево выбора - одна из форм промежуточного представления (intermediate representation). После синтаксического и семантического анализа мы генерируем более низко-уровневое промежуточное представление, которое будет иметь вид машинно-подобных инструкций.

Пример:

```
value = pi * 3
```

Превращается в...

```
t1 = inttofloat(3)
t2 = pi * t1
x = t2
```

Каждый operand исполняют роль регистров. Каждая инструкция выполняет не более одной операции. Это промежуточное представление называется *three-address code*. С помощью такого представления можно легко записывать ассемблерный код. **Three-address code**

Оптимизации

Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.

Граф потока управления — множество всех возможных путей исполнения программы, представленное в виде графа. Вершины: действия. Рёбра: следует за.

Виды оптимизаций с конкретными примерами

- Кросс-платформенные, платформенно-независимые, platform-independent
- **Constant folding** Свёртка констант - часто используемые в современных компиляторах оптимизации, уменьшающие избыточные вычисления, путём замены константных выражений и переменных на их значения. В вышеупомянутом three-address code выражение inttofloat(3) можно заменить на число с плавающей точкой 3.0.

```
t1 = 3.0
t2 = pi * t1
x = t2
```

- **CSE - common subexpression elimination**

Удаление общих подвыражений — оптимизация компилятора, которая ищет в программе вычисления, выполняемые более одного раза на рассматриваемом участке, и удаляет вторую и последующие одинаковые операции, если это возможно и эффективно. Данная оптимизация требует проведения анализа потока данных для нахождения избыточных вычислений и практически всегда улучшает время выполнения программы в случае применения.

- **DCE - dead code elimination**

В теории компиляторов удалением **мёртвого кода** называется оптимизация, удаляющая мёртвый код. Мёртвым кодом (так же бесполезным кодом) называют код, исполнение которого не влияет на вывод программы, все результаты вычисления такого кода являются мёртвыми переменными, то есть переменными, значения которых в дальнейшем в программе не используются.

- **UCE - unreachable code elimination**

В теории компиляторов удалением **недостижимого кода** называется оптимизация, удаляющая недостижимый код, то есть код, который содержится в программе, но по каким-то причинам никогда не исполняется. В графе потока управления программы этот код содержится в узлах, недостижимых из начального узла

- **Loop unroll - раскрутка циклов**

Техника оптимизации компьютерных программ, состоящая в искусственном увеличении количества инструкций, исполняемых в течение одной итерации цикла. В результате применения этой оптимизации увеличивается количество инструкций, которые потенциально могут выполняться параллельно, и становится возможным более интенсивное использование регистров, кэша данных и исполнительных устройств.

- [Еще всякие разные](#)
 - Платформенно-зависимые — для конкретного процессора
- Распределение регистров
- Выбор оптимальной инструкции/оптимального представления

Типы оптимизаций

- **Reephole-оптимизация**

Локальные reephole-оптимизации рассматривают несколько соседних (в терминах одного из графов представления программы) инструкций (как будто «смотрит в глазок» на код), чтобы увидеть, можно ли с ними произвести какую-либо трансформацию с точки зрения цели оптимизации. В частности, они могут быть заменены одной инструкцией или более короткой последовательностью инструкций.

Например, удвоение числа может быть более эффективно выполнено с использованием левого сдвига или путём сложения числа с таким же.

- **Локальная оптимизация**

В локальной оптимизации рассматривается только информация одного [базового блока](#) за один шаг. Так как в базовых блоках нет переходов [потока управления](#), эти оптимизации требуют незначительного анализа (экономия времени и снижая требования к памяти), но это также означает, что не сохраняется информация для следующего шага.

- **Внутрипроцедурная оптимизация**

Внутрипроцедурные оптимизации — глобальные оптимизации, выполняемые целиком в рамках единицы трансляции (например, функции или процедуры). При такой оптимизации задействовано гораздо больше информации, чем в локальной, что позволяет достигать более значительных эффектов, но при этом часто требуются ресурсозатратные вычисления. При наличии в оптимизируемой программной единице [глобальных переменных](#) оптимизация такого вида может быть затруднена.

- **Оптимизация циклов**

Существует большое количество оптимизаций, применяемых к циклам. При большом количестве повторений цикла такие оптимизации чрезвычайно эффективны, так как небольшим преобразованием влияют на значительную часть выполнения программы. Поскольку циклы — весомая часть времени выполнения многих программ, оптимизации циклов существуют практически во всех компиляторах и являются самыми важными.

Например, выявив [инварианты цикла](#), иногда можно вынести часть операций из цикла, чтобы не выполнять избыточные повторные вычисления.

- **Межпроцедурная оптимизация**

Такие виды оптимизаций анализируют сразу весь исходный код программы. Большее

количество информации, извлекаемой данными методами, означает что оптимизации могут быть более эффективным по сравнению с другими методами. Такие оптимизации могут использовать довольно сложные методы, например, вызов функции замещается копией тела функции (встраивание или inline).

Генерация кода

Из промежуточного представления порождается код на целевом машинно-ориентированном языке.

Кодогенерация — часть процесса компиляции, когда специальная часть компилятора, **кодогенератор**, конвертирует синтаксически корректную программу в последовательность инструкций, которые могут выполняться на машине. При этом могут применяться различные, в первую очередь машинно-зависимые оптимизации. Часто кодогенератор является общей частью для множества компиляторов.

Задачи генератора кода

В дополнение к основной задаче — преобразованию кода из промежуточного представления в машинные инструкции — генератор кода обычно пытается оптимизировать создаваемый код теми или иными способами. Например, он может использовать более быстрые инструкции, использовать меньше инструкций, использовать имеющиеся регистры и предотвращать избыточные вычисления.

Некоторые задачи, которые обычно решают сложные генераторы кода:

- Выбор инструкций: какие инструкции использовать
- Планирование инструкций: в каком порядке размещать эти инструкции. Планирование — это оптимизация, которая может значительно влиять на скорость выполнения программы на [конвейерных процессорах](#)
- Размещение в регистрах: размещение переменных программы в регистрах процессора.

Выбор инструкций обычно выполняется рекурсивным обходом абстрактного синтаксического дерева, в этом случае сравниваются части конфигураций дерева с шаблонами; например, дерево `W:=ADD(X,MUL(Y,Z))` может быть преобразовано в линейную последовательность инструкций рекурсивной генерации последовательностей `t1:=X` и `t2:=MUL(Y,Z)`, а затем инструкцией `ADD W,t1,t2`.

Ассемблер

Ассемблер — [транслятор](#) программы из текста на [языке ассемблера](#), в программу [на машинном языке](#).

Как и сам язык, ассемблеры, как правило, специфичны для конкретной [архитектуры](#), [операционной системы](#) и варианта синтаксиса языка, поскольку работают с мнемониками машинных инструкций определённого процессора.

Ассемблирование может быть не первым и не последним этапом на пути получения

исполнимого модуля программы. Так, многие компиляторы с языков программирования высокого уровня выдают результат в виде программы на языке ассемблера, которую в дальнейшем обрабатывает ассемблер. В свою очередь, результатом ассемблирования может быть не исполняемый, а **объектный модуль**, содержащий разрозненные блоки машинного кода и данных программы, из которого (или из нескольких объектных модулей) в дальнейшем с помощью **редактора связей** (линкера) может быть получен **исполняемый файл**.

Исполняемый файл — набор инструкций, который заставляет компьютер выполнить определённую задачу. В отличие от **текстового файла**, который рассчитан на чтение человеком, исполняемый файл рассчитан на чтение (и выполнение) процессором. Каждый такой файл имеет свою структуру и особенности (см. формат файлов **ELF**, **PE**)

Под «инструкциями» традиционно понимается **машинный код**, который выполняется напрямую физическим процессором. В некоторых случаях файл, содержащий инструкции **сценария** промежуточного языка программирования (например, **байт-код**), также может считаться исполняемым.

В отличие от **компиляции** программ на **языках высокого уровня**, ассемблирование является более или менее однозначным и обратимым процессом, поскольку в языке ассемблера каждой мнемонике соответствует одна машинная инструкция, в то время как в высокоуровневых языках каждое выражение может преобразовываться в большое число различных инструкций (операция, обратная ассемблированию, называется **дизассемблированием**).

Компоновщик

Компоновщик — **инstrumentальная программа**, которая производит **компоновку** («линковку»): принимает на вход один или несколько объектных модулей и собирает из них исполняемый или библиотечный файл-модуль.

Объектный модуль — файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки **исходного кода** компилятором. Объектный файл содержит в себе особым образом подготовленный код (часто называемый **двоичным** или **бинарным**), который может быть объединён с другими объектными файлами при помощи редактора связей для получения готового **исполнимого модуля** либо библиотеки.

Объектные файлы представляют собой блоки машинного кода и данных с неопределенными адресами ссылок на данные и процедуры в других объектных модулях, а также список своих процедур и данных. Компоновщик собирает код и данные каждого объектного модуля в итоговую программу, вычисляет и заполняет адреса перекрестных ссылок между модулями. Связывание со **статическими библиотеками** выполняется редактором связей, а с операционной системой и **динамическими библиотеками** связывание выполняется при исполнении программы, после её загрузки в память.

Объектный модуль содержит таблицу символов для внешне видимых (публичных) идентификаторов.

Для связывания модулей компоновщик использует [таблицы символов](#), созданные компилятором в каждом из объектных модулей. Эти таблицы могут содержать символы следующих типов:

- *Определённые или экспортируемые* имена — функции и переменные, определённые в данном модуле и предоставляемые для использования другим модулям;
- *Неопределённые или импортируемые* имена — функции и переменные, на которые ссылается модуль, но не определяет их внутри себя;
- *Локальные* — могут использоваться внутри объектного файла для упрощения процесса настройки адресов.

Для большинства компиляторов один объектный файл является результатом компиляции одного файла с [исходным кодом](#). Если программа собирается из нескольких объектных файлов, компоновщик собирает эти файлы в единый исполняемый файл, вычисляя и подставляя адреса вместо символов, в течение *времени компоновки* (статическая компоновка) или во *время исполнения* (динамическая компоновка).

Компоновщик может извлекать объектные файлы из специальных коллекций, называемых библиотеками. Если не все символы, на которые ссылаются пользовательские объектные файлы, определены, то компоновщик ищет их определения в библиотеках, которые пользователь подал ему на вход. Обычно одна или несколько системных библиотек используются компоновщиком по умолчанию. Когда объектный файл, в котором содержится определение какого-либо искомого символа, найден, компоновщик может включить его (файл) в исполняемый файл (в случае статической компоновки) или отложить это до момента запуска программы (в случае динамической компоновки).

Работа компоновщика заключается в том, чтобы в каждом модуле определить и связать ссылки на неопределённые имена. Для каждого импортируемого имени находится его определение в других модулях, упоминание имени заменяется на его адрес.

Загрузчик программ

Загрузчик — в информатике, программа, отвечающая за загрузку исполняемых файлов и запуск соответствующих новых процессов. Обычно является частью операционной системы.

Под «инструкциями» традиционно понимается [машинный код](#), который выполняется напрямую физическим процессором. В некоторых случаях файл, содержащий инструкции [сценария](#) промежуточного языка программирования (например, [байт-код](#)), также может считаться исполняемым.

При запуске новой программы загрузчик должен:

- Считать данные из запускаемого файла.
- Если необходимо — загрузить в память недостающие динамические библиотеки.
- Заменить в коде новой программы относительные адреса и символические ссылки на точные, с учётом текущего размещения в памяти, то есть выполнить связывание

адресов.

- Создать в памяти образ нового процесса и запланировать его к исполнению.

Планирование заключается в назначении приоритетов [процессам в очереди с приоритетами](#). Самой важной целью планирования задач является наиболее полная загрузка процессора. *Производительность* — количество процессов, которые завершают выполнение за единицу времени. *Время ожидания* — время, которое процесс ожидает в очереди готовности. *Время отклика* — время, которое проходит от начала запроса до первого ответа на запрос.

Роль и принцип работы редактора связей (linker) при раздельной трансляции.

Раздельная компиляция

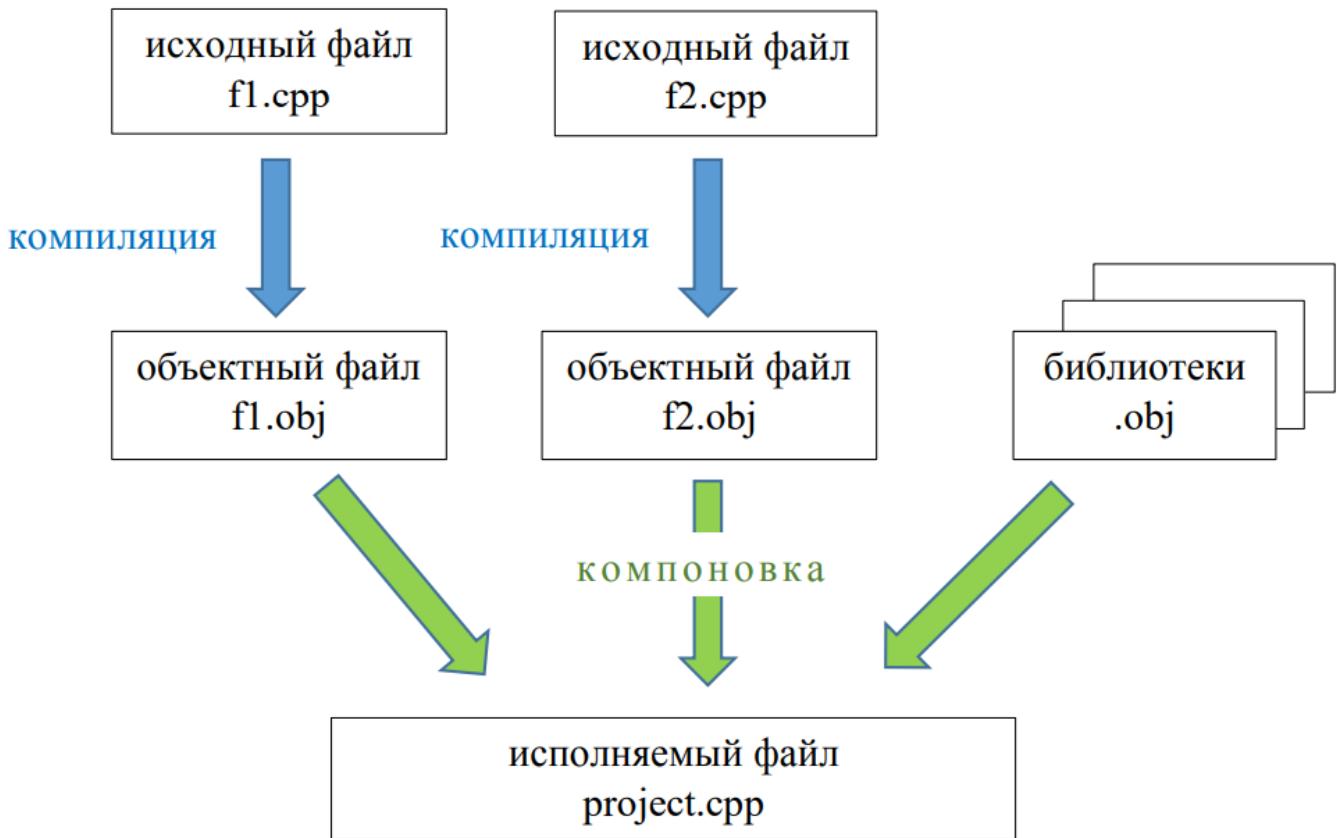
Определение

Раздельная компиляция — трансляция частей программы по отдельности с последующим объединением их [компоновщиком](#) в единый загрузочный модуль.

Смысл кратко

Механизм раздельной компиляции состоит в том, что процесс получения программы на машинном языке осуществляется в два этапа.

1. Первый этап — это компиляция. Отдельные исходные файлы компилируются независимо друг от друга. Результат компиляции одного исходного файла называется объектным модулем. В ОС Windows объектный модуль — это файл с расширением «.obj».
2. Второй этап называется компоновкой (по-английски linking). Он состоит в сборке всех объектных модулей в готовую программу на машинном языке. Кроме объектных модулей, полученных из исходных файлов программы пользователя, на этом этапе требуются дополнительные файлы, называемые библиотеками. Библиотеки содержат машинный код стандартных функций, которые используются в программе пользователя, например, математические, функции ввода-вывода. Библиотеки обычно представляют собой наборы объектных модулей, объединенных в один файл (под Windows они обычно имеют расширение .lib).



Раздельная компиляция удобна при небольших модификациях больших программ, когда имеется много отдельных файлов, и из них изменяется лишь небольшое количество. Тогда нет необходимости перекомпилировать все файлы, а можно перекомпилировать только те файлы, которые были изменены.

Однако, перекомпоновывать придется все. Смысл объединения файлов программы в проекты, например, в Visual Studio, в том и состоит, что перекомпилироваться будут только изменившиеся файлы, и каждый файл компилируется отдельно.

Небольшая историческая справка

Исторически особенностью компилятора, отражённой в его названии (*compile* — собирать вместе, составлять), являлось то, что он производил как [трансляцию](#), так и компоновку, при этом компилятор мог порождать сразу [машинный код](#). Однако позже, с ростом сложности и размера программ (и увеличением времени, затрачиваемого на перекомпиляцию), возникла необходимость разделять программы на части и выделять [библиотеки](#), которые можно компилировать независимо друг от друга. В процессе трансляции программы сам компилятор или вызываемый компилятором транслятор порождает [объектный модуль](#), содержащий дополнительную информацию, которая потом — в процессе компоновки частей в исполняемый модуль — используется для связывания и разрешения ссылок между частями программы. Раздельная компиляция также позволяет писать разные части исходного текста программы на разных языках программирования.

Появление раздельной компиляции и выделение компоновки как отдельной стадии произошло значительно позже создания компиляторов. В связи с этим вместо термина «компилятор» иногда используют термин «транслятор» как его синоним: либо в старой

литературе, либо когда хотят подчеркнуть его способность переводить программу в машинный код (и наоборот, используют термин «компилятор» для подчёркивания способности собирать из многих файлов один). Вот только использование в таком контексте терминов «компилятор» и «транслятор» неправильно. Даже если компилятор выполняет трансляцию программы самостоятельно, поручая компоновку вызываемой внешней программе-компоновщику, такой компилятор не может считаться разновидностью транслятора, — транслятор выполняет трансляцию исходной программы и только. И уж тем более не являются трансляторами компиляторы вроде системной утилиты-компилятора **make**, имеющейся во всех UNIX-системах.

Пример раздельной трансляции: утилита **make**

Собственно утилита **make** — яркий пример довольно удачной реализации раздельной компиляции. Работа утилиты **make** управляется сценарием на интерпретируемым утилитой входном языке, известном как **makefile**, содержащемся в задаваемом при запуске утилиты входном текстовом файле. Сама утилита не выполняет ни трансляцию, ни компоновку — де-факто утилита **make** функционирует как диспетчер процесса компиляции, организующий компиляцию программы в соответствии с заданным сценарием. В частности в ходе компиляции целевой программы утилита **make** вызывает трансляторы с языков программирования, транслирующие разные части исходной программы в объектный код, и уже после этого вызывается тот или иной компоновщик, компонующий конечный исполняемый программный или библиотечный программный модуль. При этом разные части программы, оформленные в виде отдельных файлов исходного текста, могут быть написаны как на одном языке программирования, так и на разных языках программирования. В процессе перекомпиляции программы транслируются только изменённые части-файлы исходного текста программы, вследствие чего длительность перекомпиляции программы значительно (порой на порядок) сокращается.

Разница между объявлением и определением

Определение связывает имя с реализацией, что может быть либо кодом либо данными:

- Определение переменной побуждает компилятор зарезервировать некоторую область памяти, возможно задав ей некоторое определённое значение.
- Определение функции заставляет компилятор сгенерировать код для этой функции

Определение содержит имя символа, его тип и адрес.

Объявление говорит компилятору, что определение функции или переменной (с определённым именем) существует в другом месте программы, вероятно в другом С файле. (Заметьте, что определение также является объявлением — фактически это объявление, в котором «другое место» программы совпадает с текущим). Объявление содержит имя символа и его тип.

Для переменных существует определения двух видов:

- **глобальные переменные**, которые существуют на протяжении всего жизненного цикла программы («статическое размещение») и которые доступны в различных функциях;

- **локальные переменные**, которые существуют только в пределах некоторой исполняемой функции («локальное размещение») и которые доступны только внутри этой самой функции.

При этом под термином «доступны» следует понимать «можно обратиться по имени, ассоциированным с переменной в момент определения».

Про редактор связей

Определение

Компоновщик — **инструментальная программа**, которая производит **компоновку** («линковку»): принимает на вход один или несколько объектных модулей и собирает из них исполняемый или библиотечный файл-модуль.

Содержание объектного файла — в сущности две вещи:

- **код**, соответствующий определению функции в С файле
- **данные**, соответствующие определению **глобальных** переменных в С файле (для инициализированных глобальных переменных начальное значение переменной тоже должно быть сохранено в объектном файле).

Код и данные, в данном случае, будут иметь ассоциированные с ними имена — имена функций или переменных, с которыми они связаны определением.

Где бы код ни ссылался на переменную или функцию, компилятор допускает это, только если он видел раньше объявление этой переменной или функции. Объявление — это обещание, что определение существует где-то в другом месте программы.

Работа компоновщика проверить эти обещания. Однако, что компилятор делает со всеми этими обещаниями, когда он генерирует объектный файл? По существу компилятор оставляет пустые места. Пустое место (ссылка) имеет имя, но значение соответствующее этому имени пока не известно.

Задачи

1. Разрешение ссылок: редактор связей проверяет, что все ссылки на функции и переменные в объектных файлах разрешены, то есть что все необходимые функции и переменные существуют и доступны для использования.
2. Обединение: редактор связей объединяет все объектные файлы в единый исполняемый файл или динамическую библиотеку.
3. Разрешение конфликтов: если в разных объектных файлах используются одинаковые имена функций или переменных, редактор связей разрешает конфликты, выбирая правильную версию для использования.

Принцип работы связывания объектных файлов

Таблица символов — это структура данных, которая содержит информацию об

объявленных и определенных символах в программе. Каждый элемент таблицы символов содержит имя символа, его тип и адрес в памяти, где он определен.

Для связывания модулей компоновщик использует [таблицы символов](#), созданные **компилятором** в каждом из объектных модулей. Эти таблицы могут содержать символы следующих типов:

- **Определённые или экспортируемые имена** — функции и переменные, определённые в данном модуле и предоставляемые для использования другим модулям;
- **Неопределённые или импортируемые имена** — функции и переменные, на которые ссылается модуль, но не определяет их внутри себя;
- **Локальные** — могут использоваться внутри объектного файла для упрощения процесса настройки адресов.

Редактор связей использует таблицу символов для связи объявлений и определений символов. Если символ объявлен в одном модуле и определен в другом, редактор связей использует таблицу символов, чтобы найти определение символа и заменить ссылку на его адрес в памяти.

Если символ объявлен и определен в одном модуле, редактор связей просто добавляет его в таблицу символов и использует его определение для разрешения ссылок на него в других модулях.

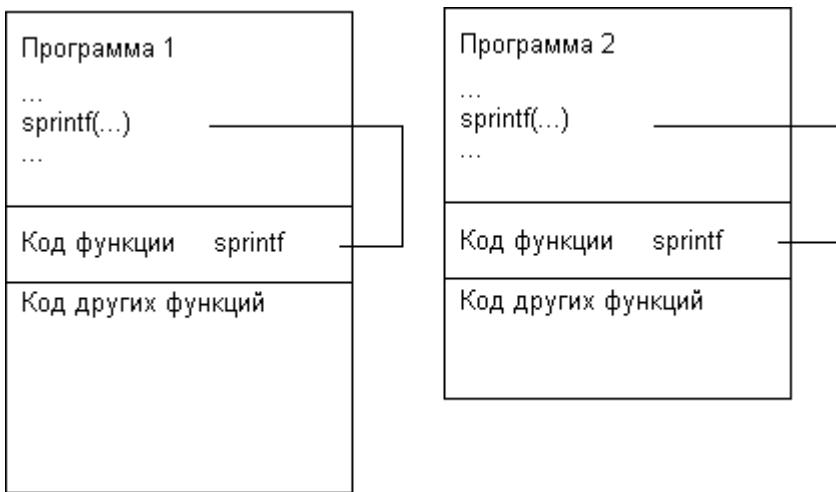
То есть в итоге получаем следующую последовательность действий:

1. Получает список всех объектных файлов, которые необходимо связать.
2. Анализирует каждый объектный файл и строит таблицу символов, которая содержит информацию обо всех используемых функциях и переменных.
3. Проверяет, что все ссылки на функции и переменные разрешены, и разрешает конфликты, если они возникают.
4. Объединяет все объектные файлы в единый исполняемый файл или динамическую библиотеку и создает таблицу символов для нового файла.
5. Выполняет финальную обработку исполняемого файла, добавляя информацию о загрузке и другую дополнительную информацию.

Принцип работы связывания с библиотеками

Связывание с библиотеками происходит похожим на связывание объектных файлов образом. Когда вы используете функции из библиотеки, компилятор создает ссылки на эти функции в вашем коде. При связывании линкер ищет эти функции в библиотеке и добавляет их в ваш исполняемый файл или динамическую библиотеку.

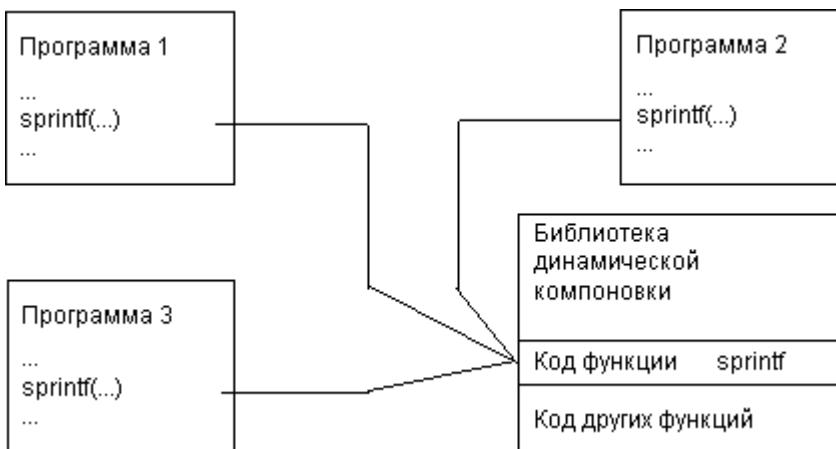
Есть два типа библиотек: статические и динамические. Статические библиотеки связываются с исполняемым файлом во время компиляции, в результате чего все функции из библиотеки копируются в исполняемый файл. Это означает, что вам не нужно предоставлять дополнительные файлы при запуске программы, но размер исполняемого файла может быть большим.



source: https://www.frolov-lib.ru/books/bsp/v27/ch3_1.htm

Статическая компоновка. Пример.

Динамические библиотеки связываются с исполняемым файлом во время выполнения программы. Это означает, что вы можете использовать одну библиотеку для нескольких программ, и изменения в библиотеке автоматически отразятся на всех программах, использующих ее. Однако, для работы программы, необходимо наличие динамической библиотеки на компьютере пользователя.



source: https://www.frolov-lib.ru/books/bsp/v27/ch3_1.htm

Динамическая компоновка. Пример.

В обоих случаях, линкер ищет библиотеки в определенных местах на вашем компьютере, таких как стандартные директории, указанные в переменной окружения PATH, или директории, указанные в параметрах линкера. Если линкер не может найти нужную библиотеку, он выдает ошибку.

Данные. Представление данных в памяти ЭВМ: целые числа (3 способа представления чисел со знаком). Двоично-десятичные числа. Арифметика с насыщением.

Данные.

Нет однозначного определения к понятию/слову **данные**. Разные стандарты, источники предлагают разные определения, среди них:

Данные — формы представления информации, с которыми имеют дело информационные системы и их пользователи (ISO/IEC 10746-2:1996).

□ **Данные** — поддающееся многократной интерпретации представление информации в формализованном виде, пригодном для передачи, связи, или обработки (ISO/IEC 2382:2015).

Any sequence of one or more symbols given meaning by specific act(s) of interpretation (англ. Википедия).

Представление целых чисел в памяти ЭВМ.

В памяти компьютера все данные, грубо говоря, представимы в виде нулей и единиц(в большинстве современных компьютерах это абстракция над наличием и отсутствием тока). Для представления чисел в ЭВМ обычно используются битовые наборы. Битовый набор - это последовательность из нулей и единиц, которая имеет фиксированную длину. Организовать обработку наборов фиксированной длины технически легче, чем наборов переменной длины. Позиция в битовом наборе называется разрядом.

Итак, число у нас хранится в памяти. Перед тем, как мы начнем выполнять с числом какие-то операции, оно будет записано в какой-то регистр процессора, который, как известно имеет ограниченное количество бит \Rightarrow записать мы можем настолько большое число, насколько это позволяет размер регистра. Из комбинаторики, мы знаем, что если N - размер регистра в битах, то 2^N - количество значений которые мы можем записать в регистр, а 2^{N-1} - максимальное число, которое в него поместится. Дальше в этом разделе, посвященном представлению целых чисел, будем считать, что размер нашего регистра равен 4 бит для большей наглядности.

Беззнаковые числа.

Если поразмышлять, как можно представлять целые числа без знака, на ум могут прийти разные идеи. Например, можно считать, что представляемое число равно количеству единиц в битовом наборе. Однако такой способ будет очень неэкономичен по памяти - одним байтом мы сможем представить максимум число 8.

Самым удобным, экономным и используемым способом представления беззнаковых чисел оказался следующий: битовый набор, соответствующий числу, является k -разрядной записью этого числа в двоичной системе счисления. Простыми словами, переводим число из десятеричной системы счисления в двоичную и получаем вид числа, в котором оно представимо в памяти ЭВМ.

$$9 \Rightarrow 1001 \Rightarrow 00001001$$

В начале число в десятеричной системе счисления, дальше в двоичной, а дальше в том виде, в котором оно представлено в ЭВМ(вспоминаем, что минимально адресуемая ячейка памяти - 8 бит).

Числа со знаком.

Представить целые знаковые числа уже более проблематично, чем представить беззнаковые. Как уже выше было сказано, данные представляются только в виде нулей и единиц, для **минуса** перед числом нет никаких символов, обозначений. Поэтому нужно было создавать какой-то стандарт, метод записи целых чисел со знаком.

Сейчас целые числа могут быть представлены тремя основными способами: **прямой код**, **обратный код**, **дополнительный код**(являющийся в данный момент основным).

Положительные числа.

Стоит сразу оговорить, что во всех трёх способах положительные целые числа представляются одинаково: старший бит, называемый битом знака, всегда равен нулю. Дальше располагается двоичная запись абсолютной величины числа. Проще говоря, это тоже самое, что и для беззнаковых чисел, только старший бит всегда равен нулю. Потеряв старший бит, мы теперь можем записать максимум 2^{N-1} значений, а максимальное число будет равно $2^{N-1} - 1$, где N - размер регистра.

Прямой код.

В качестве первого решения проблемы записи целых чисел со знаком первым был представлен прямой код - способ записи, в котором самый старший бит отводится под знак и равен 0, если число положительное и 1, если число отрицательное. Дальше идёт абсолютное значение числа в двоичной системе счисления.

$$9 \Rightarrow 1001 \Rightarrow \overline{0}0001001$$

$$-9 \Rightarrow \overline{1}0001001$$

Думаю тут всё понятно, мы берём девятку, переводим её в двоичную систему счисления и дальше если перед ней был минус в старший бит ставим единичку, если нет - ноль.

Однако, как оказалось дальше, прямой код принёс больше проблем, чем решил.

□ Проблемы прямого кода.

1. Всего диапазон положительных чисел составляет 2^{N-1} значений.

- Появляется число 1000, которое будет читаться как -0 (не забываем, что мы договорились, что число бит в нашем регистре равно 4).
- Теперь нам нужен некоторый вычитатель, который позволил бы нам отнимать числа. Представить $x - y$ как $x + (-y)$ мы не можем, так как в результате получится полнейший бред.

Вот для примера картинка, с ней будет легче осознать, что написано выше.

- **Картина**

7	0	1	1	1
6	0	1	1	0
5	0	1	0	1
4	0	1	0	0
3	0	0	1	1
2	0	0	1	0
1	0	0	0	1
0	0	0	0	0
-0	1	0	0	0
-1	1	0	0	1
-2	1	0	1	0
-3	1	0	1	1
-4	1	1	0	0
-5	1	1	0	1
-6	1	1	1	0
-7	1	1	1	1

Обратный код.

Спустя некоторое время, появляется новый способ представления отрицательных чисел, который смог разрешить некоторые из проблем. Этот способ получил название **обратный код**. Получается он следующим способом: мы берём число, представленное в прямом коде и инвертируем все биты абсолютной части числа. То-есть бит знака остается неизменным, остальные биты меняем: $0 \Rightarrow 1$, $1 \Rightarrow 0$.

$$9 \Rightarrow 1001 = 00001001$$

$$-9 \Rightarrow -1001 = 10001001 = 11110110$$

Обратный код помог нам решить проблему вычитания чисел. Теперь мы можем заменить операцию вычитания на операцию сложения, получив при этом нормальный результат. Стоит также понять, что компьютер при сложении и вычитании по прежнему продолжает воспринимать числа записанные в обратном коде, как обычные целые положительные числа. Тем самым, когда компьютеру надо выполнить операцию, например, $2 - 3$, он выполняет операцию $2[0010] + (-3)[1100]$, где (-3) он видит как двоичное представление числа $12[1100]$. В результате мы получаем число $-1[1110]$ в обратном коде.

□ **Нерешенные проблемы.** 1. Проблема отрицательного нуля. 2. Если мы попробуем по складывать другие пары чисел, заметим что результат будет неверным, он будет отличаться от истины на единичку. Например, $-3[1100] - 2[1101]$ дадут нам в результате $-6[1001]$ или $2[0010] - 1[1110]$ дадут нам в результате $0[0000]$.

- Картинка для понимания

7	0	1	1	1	
6	0	1	1	0	
5	0	1	0	1	
4	0	1	0	0	
3	0	0	1	1	
2	0	0	1	0	
1	0	0	0	1	
0	0	0	0	0	
-0	1	1	1	1	15
-1	1	1	1	0	14
-2	1	1	0	1	13
-3	1	1	0	0	12
-4	1	0	1	1	11
-5	1	0	1	0	10
-6	1	0	0	1	9
-7	1	0	0	0	8

Дополнительный код.

Следом за обратным кодом появился новый способ записи целых чисел - дополнительный код. Получается он образованием обратного кода с последующим прибавлением единицы к его младшему разряду.

Благодаря данному способу, число $-0[1111]$ исчезает ($1111 + 1 = 10000$ - превращается в обычный 0). Отрицательные числа (в нашем случае от -1 до -7) теперь записываются в порядке убывания; если сравнивать их побитово, то число -1 больше числа -2 ; число -2 больше числа -3 и т.д.

За счет того, что мы избавились от числа -0 , у нас появляется свободное место для ещё одного отрицательно числа.

При выполнении операции вычитания, решается проблема с недостающей единицей: все операции вычитания выполняются корректно. Например: $2[0010] - 1[1111]$, компьютер будет представлять как $2[0010] + 15[1111]$. В результате мы получаем число в двоичной записи из 5 бит, старший из которых отсекается: $1|0001 \Rightarrow 0001$ - получили число 1.

- Картинка для понимания

7	0	1	1	1	
6	0	1	1	0	
5	0	1	0	1	
4	0	1	0	0	
3	0	0	1	1	
2	0	0	1	0	
1	0	0	0	1	
0	0	0	0	0	
-1	1	1	1	1	15
-2	1	1	1	0	14
-3	1	1	0	1	13
-4	1	1	0	0	12
-5	1	0	1	1	11
-6	1	0	1	0	10
-7	1	0	0	1	9
-8	1	0	0	0	8

Давайте подробнее разберёмся как работает дополнительный код.

Примерчик от Луцива: представьте механический счетчик, считающий количество прошедший через него воды. Всего у нашего счетчика 6 ячеек и он может максимум показать число 999999. А что будет дальше? Правильно $1000000 = 000000$ (о переполнении будет написано ниже). Получается, нас счетчик может рассматриваться как что-то, что считает по модулю 1000000. Но какое ещё есть целое число, лежит в одном классе эквивалентности с 999999 по модулю 1000000. Верно, -1. А что если у нас счетчик стоит на нуле и мы прокачаем его в обратную сторону(сделаем -1)? Думаю очевидно, что будет 999999. $999999 \bmod (1000000) = -1$. Получается, что 999999 - дополнительный код -1 в десятеричной системе счисления, с ограничением в 6 ячеек.

Теперь поняв предыдущий пример, попробуем определить дополнительный код более формально. Пусть B - основание системы счисления, N - число разрядов (например регистра; аналогия с примером - число ячеек). Возьмём $P = B^N$. Дальше делаем наши вычисления по модулю P :

Берем число $x \in [0, P)$. Тогда $[x]_p = [x + nP]_p = \{x + Pk | k \in Z\}$ - класс эквивалентности.

Дальше берётся число $s \in [0, P - 1)$ и принимается $[-s] = [P - s]_p$.

Пример: для 16 битных чисел берётся обычно $s = \frac{2^{16}}{2} = 32768$. Тогда наш диапазон чисел (от 0 до $2^N - 1$): $[0, 65536]$ можно разбить на два диапазона: $[-32768, 0) \cup [0, 32768)$. И теперь мы просто считаем, что числа, которые больше 32768 являются отрицательными и их значение равно $32768 - P$.

Теперь ещё формально разберём как работает вычитание: $x - y = x + [-y]$. Как уже говорилось выше, $[-y] = [P - y]$, а теперь делаем самый крутой трюк - прибавим и отнимем единичку $= [(P - 1 - y) + 1]$. Почему то, что у нас получилось - действительно верный результат? Смотрите, $P - 1$ это идущие подряд единички во всех разрядах (аналогия с 999999 выше). Дальше мы отнимаем от идущих подряд единичек наш y , то есть просто инвертируем его биты. А затем прибавляем единичку и получаем дополнительный код.

- **Переполнение.**

Как следствие, в дополнительном коде, при сложении больших положительных чисел или совсем маленьких отрицательных чисел, могут возникнуть ситуации, которые могут сначала удивить. Например, для нашего случая, когда $k = 4 : 7[0111] + 3[1010]$ мы получим число $-6[1010]$. Эта ситуация, когда при сложении чисел с одним знаком, мы получаем число с другим знаком, называется переполнением. Мы просто вышли на грани возможных чисел, которые могут поместиться в наш битовый набор.

Двоично-десятичные числа.

Двоично-десятичный код (binary-coded decimal, BCD) - форма записи рациональных чисел, когда каждый десятичный разряд числа записывается в виде его четырёхбитного двоичного кода. Например, десятичное число 31_{10} будет записано в двоичной системе счисления в двоичном коде как 100110111_2 , а в двоично-десятичном коде как 001100010001_{BCD} , так как $3_{10} = 0011_2$, $1_{10} = 0001_2$. При помощи четырех бит можно закодировать шестнадцать цифр. Из

них используются 10. Остальные 6 комбинаций в двоично-десятичном коде являются запрещенными.

Преимущества:

- Для дробных чисел (как с фиксированной, так и с плавающей запятой) при переводе в человекочитаемый десятичный формат и наоборот не теряется точность.
- Упрощены умножение и деление на 10, а также [округление](#).
- Простоты для понимания. Двоично-десятичные числа это все ещё десятичные числа, просто изображенные иначе.

Недостатки:

- Требует больше памяти.
- Усложнены арифметические операции. Так как в 8421-BCD используются только 10 возможных комбинаций 4-битового поля вместо 16, существуют запрещённые комбинации битов: 1010(10), 1011(11), 1100(12), 1101(13), 1110(14), 1111(15).

Арифметика с насыщением.

Циклическая арифметика. Если результат операции выходит за пределы допустимого диапазона, то **лишние** старшие биты результата отбрасываются.

Арифметика с насыщением - это разновидность арифметики, в которой все операции ограничены фиксированным диапазоном между минимальным и максимальным значениями. Если результат оказался вне допустимого диапазона, то он считается равным граничному значению диапазона. «Насыщенные» арифметические операции доступны на многих современных платформах, и в частности были одним из расширений, сделанных на технологии Intel MMX специально для приложений обработки сигнала.

Пример.

Если допустимый диапазон значений составляет от -100 до 100, следующие арифметические операции с насыщением дают следующие значения:

Арифметика с насыщением для целых чисел также была реализована в программном обеспечении для ряда языков программирования, включая C, C++. Арифметика с насыщением позволяет создавать эффективные алгоритмы для многих задач, особенно в цифровой обработке сигналов. Например, регулировка уровня громкости звукового сигнала могут привести к переполнению, и использование арифметики с насыщением вызывает значительно меньшие искажения в звуке. Кроме того, такой тип арифметики используется в формате IEEE-754 (15 вопрос - числа с плавающей запятой), в котором переполненные значения превращаются в **бесконечность** или **минус бесконечность**, и любые другие операции с этими результатами продолжают возвращать то же самое значение.

Представление данных в памяти ЭВМ: представление вещественных с плавающей точкой, специальные значения (NaN, Inf).

(в соответствии с IEEE 754)

Введение.

Вещественными числами в компьютерной технике называются числа, имеющие дробную часть. При их изображении во многих языках программирования вместо запятой принято ставить точку. Так, например, число 5 — целое, а числа 5.1 и 5.0 — вещественные.

Как и любое другое число, вещественное число должно храниться в памяти компьютера в виде двоичного числа. Но нам нужно как-то хранить запятую, для которой нет специального обозначения - данные можно представить только в виде нулей и единиц. Здесь, как и с отрицательными числами, нужна какая-то специальная форма хранения, в которую мы сможем перевести исходное число и из которой получить его обратно.

Решать эту проблему хранения вещественных чисел взялся институт IEEE, задачей которого было разработать некоторую модель, который бы придерживались большинство разработчиков программного обеспечения. В итоге появился стандарт **IEEE-754**, описывающий способ хранения вещественных чисел(чисел с дробной частью) в памяти компьютера.

Сопроцессор.

Изначально, чтобы x86 процессоры могли выполнять операции над такими числами, начиная с 86 по 386 модель был добавлен отдельный модуль, который получил название **сопроцессор**. В более поздних версиях этот модуль был интегрирован в CPU. После появления ряда технологий от компаний Intel и AMD, появилась возможность работать с числами с плавающей точкой без сопроцессора. Однако, в современных x86 процессорах он по прежнему поддерживается, для совместимости со старыми программами или вычислениями с расширенной точностью.

Как переводить вещественные числа в двоичную систему счисления?

Пусть у нас есть число - 7.25. Разобьем его на целую и дробную часть - 7 и 25. Целую часть переводим по обычным правилам, а дробную умножаем последовательно на 2, до тех пор, пока в ответе у дробной части не получится 0. У каждого ответа мы берём только дробную часть и продолжаем умножение. В конце, то, что осталось в целой части, мы склеиваем и получаем дробную часть числа представленную в 2-м виде. Для 7.25 мы получим число 111.01.

Экспоненциальная запись числа.

Экспоненциальная запись - способ представления вещественных чисел в виде мантиссы и порядка $N = (-1)^S * M * P^E$, где N -число, S - знак, M - мантисса, P - основание системы счисления, E - порядок/экспонента.

- **Пример.**

Представим число 756.3 в экспоненциальной записи: $(-1)^2 * 7.563 * 10^2$, где 7.563 - мантисса, 10 - основание системы счисления, 2 - порядок. Или можно записать так $(-1)^2 * 75.63 * 10^1$ или $(-1)^2 * 0.7563 * 10^3$ - это все числа записанные в экспоненциальной форме.

Нормальной формой числа, называется такая экспоненциальная форма, в которой мантисса (без учёта знака) в десятичной системе находится на полуинтервале $[0;1)$.

- **Пример.**

756.3 - снова наше число. $(-1)^2 * 0.7563 * 10^3$ - нормальная форма, так как $0 < 0.7563 < 1$. А вот $(-1)^2 * 7.563 * 10^2$ - не в нормальной форме, так как $7.563 > 1$.

Проблемы нормальной формы: одно и тоже число может быть представлено кучей разных способов: $(-1)^2 * 0.7563 * 10^3$, $(-1)^2 * 0.07563 * 10^4$, $(-1)^2 * 0.007563 * 10^5$ и так далее.

Нормализованной формой числа, называется такая экспоненциальная форма, в которой в десятичной системе мантисса находится на полуинтервале $[1;10)$ (а в общем случае, если P - основание системы счисления, то в интервале $[1, P)$).

Стандартная инженерная нормализация - $M \in [1, Q)$ и отдельное обозначение нуля.

- **Пример.**

Снова наше 756.3 в десятеричной системе счисления.

$(-1)^2 * 7.563 * 10^2$ - нормализованная форма, так как $1 < 7.563 < 10$.

$(-1)^2 * 0.7563 * 10^3$ - не нормализованная форма, так как $0.7563 < 1$.

$(-1)^2 * 75.63 * 10^1$ - не нормализованная форма, так как $75.63 > 10$.

Числа с плавающей точкой. Стандарт IEEE-754.

Стандарт IEEE-754 предлагает хранить вещественное число в нормализованном виде, имеющий следующий вид: $(-1)^S * 1.M * 10^E$, где S - знак числа, M - мантисса, E - экспонента/порядок. Именно эти три числа и будут храниться в памяти ЭВМ. Иначе говоря, любое двоичное число, которое получилось в результате перевода, мы подгоняем под эту форму: то-есть сдвигаем точку до тех пор, пока слева от неё не останется только одна единица. Например, для двоичного числа $111.01 = 1.1101 * 2^2$ (двоичная система счисления, поэтому основание - двойка).

Число с плавающей запятой (или **число с плавающей точкой**) — экспоненциальная

форма представления вещественных чисел, в которой число хранится в виде мантиссы и порядка (показателя степени). То-есть это один из возможных способов представления действительных чисел, его можно считать аналогом экспоненциальной записи чисел, но только в памяти компьютера. При этом число с плавающей запятой имеет фиксированную относительную точность и изменяющуюся абсолютную.

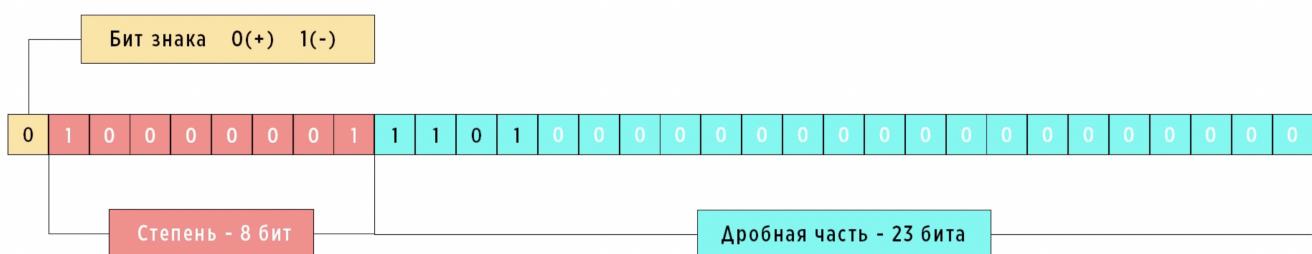
Для сохранения такого числа, стандарт предлагает нам несколько форматов, основные из которых являются форматы с одинарной(float), двойной(double / long double) и четырехкратной(long double) точностью, что соответствует 32, 64 и 128 битам. Также используется 80 битный формат расширенной двойной точности (long double). Все они одинаковы по способу хранения данных, но разные, по количеству бит, которые предоставляются для хранения.

Важно! Так как целая часть двоичного числа представленного в нормализованном виде всегда будет равна 1, она вообще не хранится в памяти, чтобы не занимать лишнее место, тем самым повышая точность нашего числа на 1 бит. Тоже самое касается основания, которое возводится в степень E.

32 - битный формат.

32 бита разбиваются на три части:

- Бит знака (1 бит).** Число **0** означает, что число положительно, число **1**, что число отрицательно.
- Мантисса (23 бит),** расположившаяся в младшей части 32 битов. Сохраняется она относительно этих 23 бит с левой стороны, все остальные свободные биты заполняются нулями.
- Порядок (8 бит),** располагает между знаком и дробной частью. Чтобы сохранить порядок(степень), надо как-то учесть её знак, потому что она может быть как положительной, так и отрицательной, а дополнительного бита для хранения знака порядка стандарт не предусматривает.



source: <https://youtu.be/U0U8Ddx4TgE?t=624>

Хранить порядок в дополнительном коде не получится, так как будут не верны результаты сравнения.

В итоге, решили хранить как положительные, так и отрицательные степени относительно числа 127. То-есть взяли 8-битные диапазон чисел от 0 до 255, в середине которого расположено число 127. Затем, сместили степень числа либо в меньшую, либо большую сторону, относительно этого числа, путем сложения степени с числом 127.

Например: $-2 + 127 = 125 < 127 \Rightarrow$ степень отрицательная, $2 + 127 = 129 > 127 \Rightarrow$ степень положительна.

Точность вычислений: 23 бита мантиссы +1 неявный (мы договорились, что целая часть - единичка всегда одинакова, поэтому её не храним) $\rightarrow \text{eps} \sim \frac{1}{2^{24}} =$

$5.96 * 10^{-8}$. Точность составляет приблизительно 7 значащих цифр.

64-битный и 128-битный форматы.

Идентично 32-битному формату, за исключением того, что под порядок отводится уже 11 бит, а под мантиссой 52 бита для 64-битного формата и 15 бит для порядка, 112 бит под мантиссой для 128-битного формата.

Точность вычислений для 64-битного формата: $\text{eps} \sim \frac{1}{2^{53}} = 1.11 * 10^{-16}$. Точность вычислений составляет приблизительно 15 значащих цифр.

Точность вычислений для 128-битного формата: $\text{eps} \sim \frac{1}{2^{113}} = 1.92 * 10^{-34}$. Точность вычислений составляет приблизительного 33 значащих цифр.

Представление специальных значений.

Максимальное число в диапазоне, которое нам доступно для сохранения порядка числа в 32 битном формате на самом деле не 255, а 254.

Дело в том, что стандартом предусмотрено хранение специальных значений, которые отличаются от обычных чисел тем, что у них все биты порядка заполнены единицами (то есть то самое число 255).

- Если в знаке стоит 0, в степени стоят все единицы, а мантисса заполнена нулями - это плюс бесконечность.
- Если в знаке стоит 1, в степени стоят все единицы, а мантисса заполнена нулями - это минус бесконечность.
- Если в знак любой, в степени все единицы, а мантисса - не нулевая, то это *Nan*

□ Как получить бесконечность? 1. Переполнение. 2. Деление не нулевого числа на ноль.

Как получить *Nan*?

$$+\infty + (-\infty)$$

$$0 * \infty$$

$$\frac{\pm 0}{\pm 0} = \frac{\pm \infty}{\pm \infty}$$

$$\sqrt[2]{x}, x < 0.$$

Так же стандарт предусматривает хранение отрицательного нуля.

Потеря точности.

Лишь некоторые из вещественных чисел могут быть представлены в памяти компьютера точным значением, в то время как остальные числа представляются приближенными значениями. Связано это с тем, что большинство вещественных чисел не представляются конечным числом нулей и единиц в двоичной системе счисления. Их подсчет идет до бесконечности и все, что нам остается сделать - это записать данное число в периоде. Получается, что есть дроби, представимые в десятичном виде конечным числом, которые невозможно представить в двоичном виде конечным числом.

Как можно бесконечность уместить в $\frac{23}{52}$ бита? Только отсечением части числа, когда в биты мантиссы мы записываем первые $23/52/\dots$ бита, то-есть максимум, который может уместиться. Это приведет к потере точности, из-за чего восстановить исходный вид нам не удастся. При переводе двоичного числа в исходный вид в десятичной системе счисления, мы получим лишь приближенное к исходному числу.

Денормализованные числа.

Денормализованное число - вид чисел с плавающей точкой, определенный в стандарте IEEE 754. При записи в форматах float, double, long double их экспонента будет записана как 0. Для получения их значения не требуется использование неявной единицы; мантисса просто умножается на наименьшую для данного формата экспоненту. То-есть, мантисса теперь начинается теперь не с единицы, а с нуля, а порядок - минимально возможный. Денормализованные числа находятся ближе к 0, чем наименьшее представимое нормализованное число.

Машинный 0 — также денормализованное число.

При записи в форматах float и double в поле порядка будет записан 0.

- Кому ничего не понятно - источники

Большинство материала взято из видео, в котором все разжевано: <https://www.youtube.com/watch?v=U0U8Ddx4TgE> https://ru.wikipedia.org/wiki/Денормализованные_числа
https://ru.wikipedia.org/wiki/IEEE_754-2008

Арифметика с фиксированной точкой. Точность и ошибки в вычислениях в сравнении с float/double.

Определение

Число с фиксированной запятой (точкой) — формат представления **вещественного числа** в памяти **ЭВМ** в виде **целого числа**. Грубо говоря, это метод представления дробных (нечелых) чисел путем хранения фиксированного количества цифр их дробной части. При этом само

число x и его целочисленное представление x' (мантиssa) связаны формулой $x = x' \cdot z$, где z — цена младшего разряда.

В представлении с фиксированной точкой дробь часто выражается в той же [системе счисления](#), что и целая часть, но с использованием отрицательных степеней z . Таким образом, если хранится f цифр дроби, значение всегда будет целым числом, кратным z^{-f} . Представление с фиксированной запятой также можно использовать для пропуска младших разрядов целочисленных значений, например, при представлении больших значений в долларах как кратных 1000 долларов.

Если не требуется, чтобы какие-либо конкретные дробные числа входили в разрядную сетку, программисты обычно выбирают $z = 2^{-f}$ ($-f$ — экспонента) — это позволяет использовать в операциях умножения и деления [битовые сдвиги](#). Про такую арифметику говорят: « f битов на дробную часть, $i = n - f$ — на целую» и обозначают как « i, f », « $i.f$ » или « $Qi.f$ ».

Например: арифметика 8,24 отводит на целую часть 8 бит и 24 — на дробную. Соответственно, она способна хранить числа от -128 до $128 - z$ ($127 + 0.111\dots111_2$) с ценой младшего разряда $z = 2^{-24} = 5, 96 \cdot 10^{-8}$.

Пример для понимания

$$2,375_{10} = 19_{10} * 2^{-3} = \overbrace{10011_2}^{n} * 2^{-3} = \underbrace{10_2}_{i} + \underbrace{011_2 * 2^{-3}}_{f} = 2_{10} + 0,375_{10}$$

экспонента $f = 3$ $z = 2^{-3}, Q2.3$

мантиssa $x' = 19$ $n = 5$ $i = 2$

То есть представление дробного числа с фиксированной точкой — это, по сути, целое число, которое должно быть неявно умножено на фиксированный коэффициент масштабирования (в английской литературе scaling factor, далее использован термин коэффициент масштабирования, чем является z). Например, значение 1,23 можно сохранить в переменной как целочисленное значение 1230 с неявным коэффициентом масштабирования $\frac{1}{1000}$ (это означает, что последние 3 десятичных разряда неявно считаются десятичной дробью).

Это представление позволяет стандартным целочисленным [арифметическим устройствам](#) выполнять вычисления с рациональными числами.

Отрицательные значения обычно представляются в двоичном формате с фиксированной точкой как целое число со знаком в [дополнительном](#) коде с неявным коэффициентом масштабирования, как указано выше.

Некоторая проблема Округления

Даже при самом тщательном округлении значения с фиксированной точкой, представленные с коэффициентом масштабирования s , могут иметь ошибку до $\pm 0,5$ в сохраненном целом числе, то есть $\pm 0,5s$ в значении. Поэтому меньшие коэффициенты масштабирования обычно дают более точные результаты.

С другой стороны, меньший коэффициент масштабирования означает меньший диапазон значений, которые могут быть сохранены в данной программной переменной. Максимальное значение с фиксированной точкой, которое может быть сохранено в переменной, равно наибольшему целочисленному значению, которое может быть сохранено в ней, умноженному на коэффициент масштабирования; и аналогично для минимального значения.

Сложение и вычитание

Чтобы сложить или вычесть два значения с одним и тем же неявным коэффициентом масштабирования, достаточно сложить или вычесть лежащие в основе целые числа; результат будет иметь свой общий неявный коэффициент масштабирования, поэтому может быть сохранен в тех же программных переменных, что и операнды. Эти операции дают точный математический результат до тех пор, пока не происходит [переполнения](#), то есть до тех пор, пока полученное целое число может быть сохранено в переменной принимающей программы. Если значения имеют разные коэффициенты масштабирования, то перед операцией их необходимо привести к общему коэффициенту масштабирования.

Пример сложения двух чисел

```
int32_t a = 0x1000L;      // q15: a = 0.125
int32_t b = 0x2000L;      // q20: b = 0.125
int32_t c = 0;            // q25
c = (a << 5) + b;        // q20: (a * 2 ^ (20 - 15) + b); c = 0x40000L (0.25 в q20)
c <= 5;                  // q25: c = 0x800000L (0.25 в q25)
```

Умножение

Чтобы умножить два числа с фиксированной точкой, достаточно умножить два основных целых числа и предположить, что коэффициент масштабирования результата является произведением их коэффициентов масштабирования. Результат будет точным, без округления, при условии, что он не переполняет принимающую переменную.

Например, умножение чисел 123 в масштабе $\frac{1}{1000}$ (0,123) и 25 в масштабе $\frac{1}{10}$ (2,5) дает целое число $123 \cdot 25 = 3075$ в масштабе $\frac{1}{1000} \cdot \frac{1}{10} = \frac{1}{10000}$, то есть $\frac{3075}{10000} = 0,3075$. В качестве другого примера, умножение первого числа на 155, неявно масштабированное на $\frac{1}{32}(\frac{155}{32} = 4,84375)$, дает целое число $123 \cdot 155 = 19065$ с неявным коэффициентом масштабирования $\frac{1}{1000} \cdot \frac{1}{32} = \frac{1}{32000}$, то есть $\frac{19065}{32000} = 0,59578125$.

Пример умножения двух чисел с последующим сложением с 3 числом

```
int32_t a = 0x8000L;           // q16: a = 0.5
int32_t b = 0x10000L;          // q21: b = 0.5
int32_t c = 0xC0000L;          // q20: c = 0.75
int64_t d;                    // Временная переменная с увеличенным числом разрядов,
чтобы хватило на результат.
d = (int64_t)a * (int64_t)b; // q37 = q16 * q21; d = 0x800000000L (0.25 in q37)
d >>= 17;                     // q37 / 2 ^ 17 = q20
c += (int32_t)d;              // q20: c = 0x10000 (1 in q20)
```

Деление

Чтобы разделить два числа с фиксированной точкой, нужно взять целое частное их основных целых чисел и предположить, что коэффициент масштабирования является частным их коэффициентов масштабирования. Как правило, первое деление требует округления, поэтому результат не является точным.

Например, деление 3456 в масштабе $\frac{1}{100}$ (34, 56) и 1234 в масштабе $\frac{1}{1000}$ (1,234) дает целое число $3456 \div 1234 = 3$ (округленное) с коэффициентом масштабирования $\frac{1}{100} / \frac{1}{1000} = 10$, то есть 30. В качестве другого примера, деление первого числа на 155 в масштабе $\frac{1}{32}$ ($\frac{155}{32} = 4,84375$) дает целое число $3456 \div 155 = 22$ (округленное) с неявным коэффициентом масштабирования $\frac{1}{100} / \frac{1}{32} = \frac{32}{100} = \frac{8}{25}$, то есть $22 \cdot \frac{8}{25} = 7,04$.

Если результат не является точным, ошибку, вызванную округлением, можно уменьшить или даже устраниТЬ, преобразовав делимое в меньший коэффициент масштабирования. Например, если r Unknown characterUnknown character = 1, 23 представлено как 123 в масштабе $\frac{1}{100}$, а s Unknown characterUnknown character = 6, 25 представлено как 6250 в масштабе $\frac{1}{1000}$, то простое деление целых чисел дает $123 \div 6250 = 0$ (округлено) с коэффициентами масштабирования $\frac{1}{100} / \frac{1}{1000} = 10$. Если r сначала преобразовать в 1230000 с коэффициентом масштабирования $\frac{1}{1000000}$, результатом будет $1230000 \div 6250 = 197$ (округлено) с коэффициентом масштабирования $\frac{1}{1000}(0,197)$. Точное значение $6,25^{1,23}$ равно 0,1968.

Пример деления двух чисел без потерь

```
int32_t a = 0x4000L;           // q15: a = 0.5
int32_t b = 0x8000L;            // q20: b = 0.5
int32_t c = 0;                  // q25
int64_t d;                    // Временная переменная с увеличенным числом разрядов.
d = (int64_t)a << 30;          // q45: d = 0x2000000000000000; (0.5 in q45)
c = (int32_t)(d / (int64_t)b); // q25: c = 0x2000000; (1 in q25)
```

Преобразование коэффициента масштабирования

- Сохранение значения в переменной, которая имеет другой неявный коэффициент

масштабирования

- Преобразование коэффициентов масштабирования двух значения для сложения или вычитания
- Восстановление исходного коэффициента масштабирования значения после умножения или деления его на другое
- Повышение точности результата деления
- Гарантия, что коэффициент масштабирования произведения или частного представляет собой простую степень, такую как 10^n или 2^n
- Сохранение результата операции в переменную без переполнения

Чтобы преобразовать число с коэффициентом масштабирования R в значение с коэффициентом масштабирования S , основное целое число должно быть умножено на отношение $\frac{R}{S}$. Таким образом, например, чтобы преобразовать значение $1,23 = \frac{123}{100}$ с коэффициентом масштабирования R в значение с коэффициентом масштабирования S $= \frac{1}{100}$, целое число 123 должно быть умножено на $\frac{1}{100} / \frac{1}{1000} = 10$, что дает представление $\frac{1230}{1000}$.

Если S не делит R (в частности, если новый коэффициент масштабирования S больше исходного R), новое целое число, возможно, придется [округлить](#).

Например, если общий коэффициент масштабирования равен $\frac{1}{100}$, умножение $1,23$ на $0,25$ влечет за собой умножение 123 на 25 , чтобы получить 3075 с промежуточным коэффициентом масштабирования $1/10000$. Чтобы вернуться к исходному коэффициенту масштабирования $1/100$, целое число 3075 затем необходимо умножить на $1/100$, чтобы получить либо $31(0, 31)$, либо $30(0, 30)$, в зависимости от [политики округления](#).

Преобразование с числами с плавающей запятой

Чтобы преобразовать число из числа с плавающей запятой в число с фиксированной запятой, можно умножить его на коэффициент масштабирования S , а затем округлить результат до ближайшего целого числа. Необходимо позаботиться о том, чтобы результат поместился в целевую переменную или регистр. В зависимости от коэффициента масштабирования и размера хранилища, а также от диапазона входных чисел преобразование может не повлечь за собой никакого округления.

Чтобы преобразовать число с фиксированной запятой в число с плавающей запятой, можно преобразовать целое число в число с плавающей запятой, а затем разделить его на коэффициент масштабирования S . Это преобразование может повлечь за собой округление, если абсолютное значение целого числа больше 2^{24} (для двоичного числа одинарной точности с плавающей запятой IEEE) или 2^{53} (для двойной точности). Переполнение или [недополнение](#) может произойти, если основное целое число соответственно очень большое или очень маленькое.

Точность и ошибки в вычислениях в сравнении с float/double.

Общие плюсы и минусы

Недостатки

- Пониженный (в простейшем случае) диапазон значений переменных по сравнению с плавающей запятой.
- Необходимость алгоритмически контролировать диапазон значений переменных. Значительная часть времени при разработке уходит на правильное масштабирование и выбор диапазонов.
- Необходимость следить за разрядностью на каждом этапе вычислений.
- Необходимость писать собственный фреймворк базовых функций (тригонометрических, логарифмических и т.п.) или модифицировать существующий.

Достоинства

- Предсказуемость результата. При правильном подходе к кодированию результат вычислений будет одинаков на любой платформе (процессор компилятор) с точностью до разряда.
- Полный контроль за поведением кода. Фиксированная точка исключает появление «неожиданностей», связанных с особенностями реализации плавающей запятой на используемой платформе.
- Автоматическая «фильтрация» пренебрежимо малых значений. В плавающей запятой ошибки вычислений могут накапливаться, в фиксированной точке этого не происходит (за счет отбрасывания малых значений) или процесс накопления ошибок можно контролировать алгоритмически.
- Алгоритмически контролируемый диапазон значений переменных.
- Переносимость алгоритмов. Разработав один раз алгоритм в фиксированной точке, портировать его на различные «слабые» платформы становится гораздо проще.
- Возможность контролировать сложность вычислений путем понижения точности при разработке алгоритма.

Сравнение с числами с плавающей запятой

- Вычисления с фиксированной запятой могут быть быстрее и/или использовать меньше оборудования, чем вычисления с плавающей запятой. Если диапазон представляемых значений известен заранее и достаточно ограничен, фиксированная точка может лучше использовать доступные биты. Например, если 32 бита доступны для представления числа от 0 до 1, представление с фиксированной точкой может иметь ошибку менее $1, 2 \cdot 10^{-10}$, тогда как стандартное представление с плавающей запятой может иметь ошибку до $596 \cdot 10^{-10}$ — потому что 9 бит тратятся впустую со знаком и показателем степени динамического коэффициента масштабирования.
- Программы, использующие вычисления с фиксированной запятой, обычно более переносимы, чем программы, использующие вычисления с плавающей запятой, поскольку они не зависят от наличия FPU. Это преимущество было особенно сильным до

того, как [стандарт IEEE с плавающей запятой](#) получил широкое распространение, когда вычисления с плавающей запятой с одними и теми же данными давали разные результаты в зависимости от производителя и часто от модели компьютера.

Во многих встроенных процессорах отсутствует FPU, потому что для целочисленных арифметических блоков требуется значительно меньше [логических вентилей](#) и они занимают гораздо меньшую площадь [микросхемы](#), чем FPU, а программная [эмуляция](#) операций с плавающей запятой на низкоскоростных устройствах была бы слишком медленной для большинства приложений. Микросхемы ЦП для более ранних [персональных компьютеров и игровых консолей](#), таких как Intel 386 и 486SX, также не имели FPU.

- Абсолютная погрешность чисел с фиксированной запятой является постоянной во всем диапазоне, а именно коэффициент масштабирования s . Напротив, относительная погрешность **чисел с плавающей запятой примерно постоянно во всем их диапазоне, варьируясь в пределах значения основания, тогда как их абсолютная погрешность варьируется на много порядков, как и сами значения.
- Во многих случаях ошибки [округления и усечения](#) при вычислениях с фиксированной точкой легче анализировать, чем при эквивалентных вычислениях с плавающей запятой. Применение методов линеаризации к усечению, таких как [дизеринг](#) и/или [формирование шума](#), более прямолинейно в рамках арифметики с фиксированной точкой. С другой стороны, использование фиксированной точки требует от программиста большей осторожности. Чтобы избежать переполнения, требуется гораздо более точные оценки диапазонов переменных и всех промежуточных значений в вычислении, а также часто дополнительный код для настройки коэффициентов масштабирования. Программирование с фиксированной точкой обычно требует использования [целочисленных типов разного размера](#).
- На почитать интересный метод — [Block floating point](#)

Применение

- Используется там, где нужно предсказуемое поведение и нужно избежать округлений чисел.
- Часто используются там, где использование чисел с плавающей запятой затратно или не поддерживается.

Например, видеосопроцессоры приставок [PlayStation](#), [Saturn](#), [Game Boy Advance](#), [Nintendo DS](#), [GP2X](#) используют арифметику с фиксированной запятой для того, чтобы увеличить пропускную способность на архитектурах без [FPU](#).

- Числа с фиксированной запятой используют там, где +известен диапазон значений и требуется производительность. В большинстве современных процессоров фиксированная запятая аппаратно не реализована, но даже программная ФЗ очень быстра — поэтому она применяется в разного рода игровых движках, растеризаторах. Но в большинстве случаев все равно используют float.

Например, [движок Doom](#) для измерения расстояний использует арифметику Q16.16, для

углов — $360^\circ = 2^{32}$.

- Также фиксированную запятую удобно использовать для записи чисел, которые по своей природе имеют постоянную **абсолютную погрешность**: координаты в **программах вёрстки, отметки времени, денежные суммы**.

Например, и сдачу в супермаркете, и налоги в стране вычисляют с точностью до сотой доли.

Другое распространенное представление

Для угловых величин зачастую делают $z = 2\pi \cdot 2^{-f}$ (особенно если тригонометрические функции вычисляются по таблице).

Материалы

- [Английская википедия](#)
- [Статья на хабре](#)
- [Русская википедия](#)
- [Видео](#)

Представление данных в памяти ЭВМ: литералы (символы и строки). Кодировки (code pages). ASCII-Z. UTF-8.

Символы

В памяти ЭВМ каждый символ закодирован в виде беззнакового целого двоичного числа. Когда компьютер читает код символа из памяти, он видит, что этот код соответствует, допустим, букве **a**. Чтобы этот код отображался одинаково на разных устройствах, все компьютеры должны использовать какой-то единый стандарт кодирования. Такой стандарт называют **таблицей кодировок символов**. Таблица кодировок символов – это соглашение об однозначном соответствии каждому символу одного беззнакового целого двоичного числа, называемого кодом этого символа.

ASCII

Изначально компьютерные технологии развивались преимущественно в США и в ходе экспериментов там был создан стандарт, подходящий для английского языка, который получил название ASCII (American Standard Code for Information Interchange).

В связи с тем, что при создании ASCII не учитывались потребности в поддержке других языков, на каждый ASCII символ было выделено всего 7 бит. Тем самым можно было закодировать 128 символов (от 0 до 127). Однако, минимальная ячейка памяти, которую можно отрисовать – 8 бит. Поэтому для хранения одного ASCII символа используется 1

байт(старший бит всегда остается пустым).

В стандарте ASCII первые 31 символ являются управляющими, остальные имеют внешний вид и отображаются на экране.

Пока программное обеспечение выпускалось преимущественно для англоязычной аудитории все было хорошо, но время шло и компьютеры распространялись по всему миру. Тут возникает главная проблема - необходимость выпускать программное обеспечение не только для англоязычной аудитории, но и для тех, кто на английском не говорит.

Кодовые страницы

Для решения данной проблемы, за счет наличия не задействованного старшего бита, было принято решение о расширении стандарта ASCII, введя кодовые страницы.

Кодовая страница - это таблица из 256 символов для конкретного языка. Первые 128 символов полностью совпадали с ASCII, остальные 128 символов были разные для каждого языка, для которого предназначалась данная кодовая страница.

В разных частях планеты начинают создавать кодовые страницы для своих языков, причем не упираясь на какие-то правила и стандарты, которых просто не было. Доходило даже до того, что в одной стране, для одного языка использовались разные кодовые страницы. Это приводило к тому, что текст, написанный на одном компьютере - был не читаем на другом, если их кодовые страницы не совпадают.

Ещё одним недостатком кодовых страниц являлась невозможность охватить всевозможные языки. Например, тысячи китайских иероглифов никак не смогли бы быть представлены лишь 128 символами. Ситуация усугубилась ещё хуже, как только появился интернет. Отсутствие мировой стандартизации привело к хаосу.

KOI8-R

KOI8-R - это 8-битная кодировка символов, полученная программистом Андреем Черновым в 1993 году и предназначенная для охвата русского языка, в котором используется кириллица. KOI8-R был основан на русском коде Морзе, который был создан из фонетической версии латинского кода Морзе.

Разработчики разместили символы русского алфавита таким образом, что позиции символы кириллицы соответствовали их фонетическим аналогам в английском алфавите в нижней части таблицы. Это означает, что если в тексте, написанном в КОИ-8, для каждого символа убрать по одному биту слева, то получится относительно читаемый текст, подобный транслиту. Например, слова «Русский Текст» превратятся в «rUSSKIJ tEKST». Из-за этого символы кириллицы расположены не в алфавитном порядке.

Такой код обмена информацией применялся в семидесятые годы на компьютерах серии ЕС ЭВМ, а с середины восьмидесятых его стали использовать в первых русифицированных версиях операционной системы UNIX.

Windows-1251.

С развитием компьютеров, активно развивались и операционные системы. На первый план вышла операционная система Windows, которая взяла за основу стандарт кодирования ASCII, дополнила его своими символами и назвала ANSI(American National Standards Institute).

На основе ANSI стали создавать кодовые страницы, под названием windows-125x, где x - номер страны(алфавита). Для кириллицы это windows-1251.

Windows-1251 выгодно отличается от других 8-битных кириллических кодировок наличием практически всех символов, использующихся в русской типографике для обычного текста (отсутствует только значок ударения); она также содержит все символы для других славянских языков: украинского, белорусского, сербского, македонского и болгарского.

Windows-1251 имеет несколько недостатков: строчная буква «я» имеет код 0xFF (255 в десятичной системе). Она является «виновницей» ряда неожиданных проблем в программах без поддержки чистого 8-го бита, а также (гораздо более частый случай) использующих этот код как служебный; отсутствуют символы псевдографики, имеющиеся в CP866 и KOI8(хотя для самих Windows, для которых она предназначена, в них не было нужды, это делало несовместимость двух использовавшихся в них кодировок заметнее); отдельное расположение буквы «ё», тогда как остальные символы расположены строго в алфавитном порядке. Это усложняет программы лексикографического упорядочения.

Операционная система Windows начинает распространяться по всему миру, дойдя до Азии, где кодовыми страницами проблему было не решить.

Unicode

В качестве единого мирового стандарта кодирования, был создан Unicode, призванный заменить ASCII.

Unicode - это стандарт кодирования символов, включающий в себя знаки почти всех письменных языков мира. Его особенность заключается в том, что если в стандарте ASCII на каждый символ выделялось 7 бит и максимальное количество символов составляло 128, то для Unicode не было никаких ограничений на количество символов. Каждый его символ представлялся минимум 2 байтами, а это уже возможность закодировать $2^{16} = 65536$ символов. При этом, в Unicode первые 128 символов совпадают с ASCII.

У каждого символа в Unicode есть описание. Например, для буквы В, это *cyrillic capital letter ve*.

Стандарт состоит из двух основных разделов: универсальный набор символов (UCS) и семейство кодировок (UTF), речь о которых пойдёт позже.

В первой версии Unicode все необходимые символы умещались в эти 2 байта. Количество символов не превышало числа 65536. В текущей версии Unicode(13) уже более 143 тысяч символов.

Для записи кода символа было принято решение использовать шестнадцатеричный

формат вида **U + 0520**(код символа).

Проблемы, к которым привёл Unicode

Unicode смог решить проблему с нехваткой символов, однако с его появлением появились и новые проблемы.

1. Любые символы в Unicode занимали минимум 16 бит, что в англоязычных странах воспринималось как расточительство памяти, когда можно было использовать 8 битный ASCII. Из-за этого многие продолжали пользоваться ASCII, игнорировав Unicode.
2. Проблема, связанная с различным разным порядком расположения байт в компьютере. Байты могут располагаться в прямой и обратной последовательности, то-есть big endian и little endian. Тем самым, при пересылки текста, с компьютера с одним порядком расположения на компьютер с другим порядком, он просто превращался в набор непонятных символов. Раньше такой проблемы не было, так как для хранения ASCII символа использовался только 1 байт и порядком расположения байтов был не важен.

Для решения этой проблемы появляются кодировки - правила описывающие хранение Unicode символов в памяти.

UCS-2

Один из первых наборов символов получил название UCS-2 (Universal coded character set). Этот набор символов хранил все символы используя фиксированную длину в 2 байта, отсюда и цифра **2** в названии. Проблема с порядком хранения байтов в данной кодировке решается следующим образом: перед отправкой или сохранением файла, в самое начало добавлялось 2-х байтовое число, которое получило название ВОМ-байты. Если использовался big endian, то эти байты шли в прямой последовательности, если little endian, то они менялись местами. Получавший компьютер считывал первые два байта и понимал в какой последовательности расположена строка.

В результате этого решения Unicode строка стала занимать ещё больше памяти, чем было.

UTF-8

В качестве универсальной кодировки, способной решить проблему как с порядком хранения байтов, так и с экономией памяти, на свет появилась UTF-8 (Unicode transformation format), которая в последствии стала доминирующей кодировкой в Интернете.

UTF-8 — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII.

Решение заключалось в том, чтобы сделать коды символов не фиксированной, а переменной длины, от 1 до 4 байт. Минимальная длина, которой может быть представлен символ - 8 бит, отсюда и цифра **8** в названии. Важным моментом является то, что первые 128 символов полностью совпадают с первыми 128 символами ASCII. Именно поэтому, эти первые 128 символов хранятся в памяти компьютера, используя один байт - то, чего все так

долго ждали. Остальные символы, занимают в памяти от 2 до 4 байт.

Как UTF-8 кодирует символы?

Существует 4 маски, для кодов разной длины. Длина первых 128 кодов помещаются в диапазон от 0 до 127, значит занимают 7 бит → старший бит принимает маску **0**. Дальше по аналогии, для хранения кода из двух байт, в первом байте применяется маска **110**, из трех байт - **1110**, из четырёх - **11110**. А в последующих байтах **10**.

Благодаря этому, также становится легко определить в какой последовательности расположены байты в памяти и необходимость в использовании ВОМ-байтов пропадает.

ВАЖНО ПОНИМАТЬ:

Стандарт ≠ кодировка.

Стандарт это просто некая **таблица** в которой указано, что вот такое число равно такой букве, например - Unicode. Кодировка же - это правила по которым записываются эти значение в память компьютера, например, UTF-8, UTF-16.

Строки

Литерал (англ. literal), или **безымянная константа** (англ. nameless constant) — запись в исходном коде компьютерной программы, представляющая собой фиксированное значение

В программировании, строковый тип — тип данных, значениями которого является произвольная последовательность (строка) символов алфавита. Некоторые языки программирования накладывают ограничения на максимальную длину строки, но в большинстве языков подобные ограничения отсутствуют. Можно представлять несколькими способами.

1. Массив символов. В этом подходе строки представляются массивом символов; при этом размер массива хранится в отдельной (служебной) области. От названия языка Pascal, где этот метод был впервые реализован, данный метод получил название *Pascal strings*.

Преимущества:

- программа в каждый момент времени содержит сведения о размере строки, поэтому операции добавления символов в конец, копирования строки и собственно получения размера строки выполняются достаточно быстро;
- строка может содержать любые данные;
- возможно на программном уровне следить за выходом за границы строки при её обработке;
- возможно быстрое выполнение операции вида «взятие N-ого символа с конца строки».

Недостатки:

- проблемы с хранением и обработкой символов произвольной длины;
- увеличение затрат на хранение строк — значение «длина строки» также занимает место и в случае большого количества строк маленького размера может существенно увеличить требования алгоритма к оперативной памяти;
- ограничение максимального размера строки. В современных языках программирования это ограничение скорее теоретическое, так как обычно размер строки хранится в 32-битовом поле, что даёт максимальный размер строки в 4 294 967 295 байт (4 гигабайта).

2. Метод «завершающего байта». Второй метод заключается в использовании «завершающего байта». Одно из возможных значений символов алфавита (как правило, это символ с кодом 0) выбирается в качестве признака конца строки, и строка хранится как последовательность байтов от начала до конца. Есть системы, в которых в качестве признака конца строки используется не символ 0, а байт 0xFF (255) или код символа «\$».

Нуль-терминированная строка или С-строка или ASCII-Z-строка — способ представления строк в языках программирования, при котором вместо введения специального строкового типа используется массив символов, а концом строки считается первый встретившийся специальный нуль-символ (NUL из кода ASCII, со значением 0).

Преимущества:

- отсутствие дополнительной служебной информации о строке (кроме завершающего байта);
- возможность представления строки без создания отдельного типа данных;
- отсутствие ограничения на максимальный размер строки;
- экономное использование памяти;
- простота получения суффикса строки;
- простота передачи строк в функции (передаётся указатель на первый символ).

Недостатки:

- долгое выполнение операций получения длины и конкатенации строк;
- отсутствие средств контроля за выходом за пределы строки, в случае повреждения завершающего байта возможность повреждения больших областей памяти, что может привести к непредсказуемым последствиям — потере данных, краху программы и даже всей системы;
- невозможность использовать символ завершающего байта в качестве элемента строки.
- невозможность использовать некоторые кодировки с размером символа в несколько байт (например, UTF-16), так как во многих таких символах, например Ä (0x0100), один из байтов равен нулю (в то же время, кодировка UTF-8 свободна от этого недостатка).

3. Использование обоих методов. Стока размещается в массиве символов определённой длины, причём её конец обозначается нулевым символом. По умолчанию, весь массив заполнен нулевыми символами. Такой способ позволяет объединить многие преимущества обоих подходов, а также избежать большинство их недостатков.

4. Представление в виде списка.

Языки Erlang, Haskell, Пролог используют для строкового типа список символов. Этот метод делает язык более «теоретически элегантным», но приносит существенные потери быстродействия.

- Кто ничего не понял; источники

По большей части взято отсюда:

[КАК РАБОТАЮТ КОДИРОВКИ | ОСНОВЫ ПРОГРАММИРОВАНИЯ](#)

Здесь изложено более подробно и за рамки того, что необходимо.

(Язык С) Типы. Представление значений в памяти ЭВМ. Адресная арифметика.

Типы данных в С

Понятие типа данных зависит от многих факторов (вычислителя, формальной модели, теории), но в данном контексте будем понимать как "размещение значения в памяти и правила выполнения операций с ними"

Из всех типов различают:

- Built-in/basic (встроенные, стандартные типы)
- Compound types (union, struct, []) - массивы, функции возвращающие void или ссылки на объекты + пункт 6.х [стандарта](#) (c++)
- Type alias (typedef), синонимы
 - `typedef void * voidptr_t`
 - `typedef int (_compare_func_t)(void _, void *)`

Простые типы

- Целочисленные (int, char, long, unsigned, enum, bool/_Bool)

Положительные числа сравниваются побитово: слева направо, а отрицательные наоборот

Для сравнения положительного и отрицательного отдельные инструкции

- Стандартные типы с указанием размера: `int8_t`, `uint32_t`, ...
 - *Появились позже*, но теперь лучше их знать и любить
- Вещественные (`double`, `float`, `long double`)

- Логические (bool/_Bool с true и false) – но их почти нет
 - C11, но можно считать и C99
 - #include <stdbool.h>
- Complex (float _Complex, ...)

Составные типы

- Массивы — **одинаковые подряд**
 - int a[10];
 - Выражения по типу a[-1] возвращают предыдущие в памяти переменные.
- Структуры — **разные одновременно**
 - Есть ещё особенные bit fields. { int a:5; int b:4; }
 - Поля лучше размещать от большего к меньшему (если делать в общем случае)
- Объединение — **разные взаимоисключающие**
 - union myunion { int x; float f; }
- Перечисление (enum) — набор intов с понятными названиями

```
enum week {Mon, Tue, Wed}; // создается "набор дней"
enum week day; // "экземпляр" набора
```

- Строки

Строки

- ASCII-Z
 - Массив char + \0
 - Коды символов
 - String literal **As is with zero**

ASCIIZ - С-строка, способ представления строк в языках программирования, при котором вместо специального строкового типа используется массив символов, а конец строки - первый встретившийся нуль-символ. (NUL из ASCII, со значением 0)

- Длина
- strlen (+см ниже)
- **strncasecmp** (сравнение строк)

Разница между strlen() и sizeof()

Sizeof():

- Унарный ОПЕРАТОР
- Работает со ВСЕМИ типами
- Может вычислять прямо во время компиляции

Strlen():

- Функция из библиотеки
- Предназначен для работы с массивами и строками
- Считает кол-во элементов до первого ascii нуля
- Проходит по всей строки (не получится во время компиляции)

Интересный пример:

```
char *str1 = "Sanjeev";
char str2[] = "Sanjeev";
printf("%d %d\n",strlen(str1),sizeof(str1));
printf("%d %d\n",strlen(str2),sizeof(str2));
```

Вывод:

- В первой строке 7 8,
 - 7 - strlen (все окей), 8 - размер типа char*, т.е. **указателя (всегда будет равен 8, какую бы строку не взяли)**
- Во второй строке: 7 8,
 - 7 - strlen, 8 - длина массива с учетом нулевого элемента (**может меняться от строки**)

Alignment

Зачем это нужно и как это ускоряет программу можно почитать [тут](#) и [тут](#)

- Компилятор лучше знает язык и целевую архитектуру
 - Как правильно (корректно)
 - Как быстрее (оптимальнее)
- #pragma pack (1) - отключает выравнивание. Используется, когда нужно сильно оптимизировать память Проблема не выравненных структур в том, что некоторым процессорам проще обращаться к объектам с четными позициями. Из-за этого инструкция **сделать что-нибудь с невыровненным объектом** разбивается на несколько других инструкций (можно посмотреть ассемблерный листинг у risc процессора), что может замедлить работу до трех раз.

- Выравнивание происходит по типу, который занимает больше всего памяти
- Также существует выравнивание по завершающему отступу
- Зависит от системы(32 или 64, получаем разный размер указателей)
- У компилятора clang есть опция -Wpadded, которая заставляет его генерировать сообщения об отверстиях выравнивания и отступах.

Способы упаковки (оптимизации памяти):

- Самый простой способ устраниить потерю памяти — изменить порядок элементов конструкции, уменьшив выравнивание. То есть: сделать все под поля, выровненные по указателю, первыми, потому что на 64-битной машине они будут 8 байт. Затем 4-байтовые целые числа; затем 2-байтовые; затем однобайтовые.
- Хотя переупорядочивание по размеру — самый простой способ избавиться от неряшливости, это не всегда правильно. Есть еще две проблемы: читабельность и локальность кеша. Программы — это не просто общение с компьютером, это общение с другими людьми. Читабельность кода важна (и особенно!), когда аудитория коммуникации — это только вы сами в будущем. Неуклюжее, механическое изменение вашей структуры может повредить читабельности. По возможности лучше переупорядочивать поля, чтобы они оставались в согласованных группах, а семантически связанные фрагменты данных располагались близко друг к другу. В идеале дизайн вашей структуры должен отражать дизайн вашей программы.
- Переупорядочивание лучше всего работает в сочетании с другими методами оптимизации ваших структур. Например, если у вас есть несколько логических флагов в структуре, подумайте о том, чтобы уменьшить их до 1-битных битовых полей и упаковать их в место в структуре, которое в противном случае было бы неаккуратным. За это вы понесете небольшой штраф за время доступа, но если он сожмет рабочий набор достаточно меньше, этот штраф будет перекрыт вашей прибылью от предотвращения промахов кеша.

Адресная арифметика

Адресная арифметика (address arithmetic) - это способ вычисления адреса какого-либо объекта при помощи арифметических операций над указателями, а также использование указателей в операциях сравнения. Адресную арифметику также называют арифметикой над указателями (pointer arithmetic).

1. Присваивание: присвоенное указателю целое число трактуется как адрес памяти.
2. Сложение, вычитание целых чисел: значение указателя изменяется на размер типа, на который он указывает, умноженное на полученное целое число.
3. Разность указателей
4. Сравнение: указатели равны, если указывают на один объект, больше или меньше, если указывают на элементы одного составного объекта в зависимости от их расположения относительно друг друга.

Адрес – номер ячейки памяти. Указатели содержат такие адреса (само значение указателя

тоже хранится в одной из ячеек).

Указатель(pointer) - переменная, диапазон значений которой состоит из адресов ячеек памяти или специального значения — нулевого адреса. Последнее используется для указания того, что в данный момент там ничего не записано. Под адресной арифметикой понимаются действия над указателями, связанные с использованием адресов памяти и появилась она как логичное продолжение идеи указателей наследованной от ассемблерных языков. В последних имеется возможность указать некое смещение от текущего положения.

```
int* p; // Если p указывает на адрес 200  
p++; // После операции сложения указывает на 200 + sizeof(int) = 204  
p--; // Обратно указывает на 200.
```

Проблемы ручного управления памятью. Способы предотвращения, средства обнаружения. Внешняя и внутренняя фрагментации свободного пространства.

Ручное управление памятью

Си заставляет нас вручную управлять памятью на куче (heap). Мы, как программисты, должны сами думать, как и сколько нам выделять памяти и не забывать ее освобождать.

API стандартной библиотеки для динамического выделения и освобождения памяти

В стандартной библиотеке Си есть несколько хорошо знакомых нам функций для ручного управления памятью

- **void *malloc(size_t size)** Входной параметр: размер памяти, которую требуется выделить. Возвращаемое значение: указатель на выделенный в куче участок памяти. Если ОС не смогла выделить память, то malloc возвращает NULL
- **void *calloc(size_t cnt, size_t size)** Выделяет память для массива из **cnt** элементов, каждый из которых занимает **size** байт памяти, и заполняет все байты выделенной памяти нулями. Если выделение прошло успешно, то возвращается указатель на меньший байт в выделенной памяти, который выравнен для любого типа объектов.
- **void realloc(void ptr, size_t size)** Изменяет величину выделенной памяти, на которую

указывает `ptr`, на новую величину, задаваемую параметром `size`. Возвращается указатель на блок памяти, поскольку может возникнуть необходимость переместить блок при возрастании его размера. В таком случае содержимое старого блока копируется в новый блок и информация не теряется. Если свободной памяти недостаточно для выделения в куче блока размером `size`, то возвращается нулевой указатель.

- `void free(void *ptr)` Входной параметр: указатель, значение которого получено из функции `malloc`. Вызов `free` на указателях, полученных не из `malloc` (например, `free(ptr+10)`) приведет к неопределенному поведению.

С большой силой приходит большая ответственность

и много проблем

Memory leak

Разработчики Си вручили нам мощные инструменты, но также и скинули все проблемы управления памятью на нас(

Глянем на пример:

```
int *p = (int *)malloc(100);
p = (int *)malloc(200);

// В чем проблема?
```

А в том, что сделав второй `malloc`, мы потеряли указатель, полученный в первом `malloc`.

Чем это чревато? Тем, что мы теперь не сможем отдать системе 100 байт, которые мы так легко потеряли на ровном месте. Что ж, это печально.

Это совсем простой пример **утечки памяти (memory leak)**. Утечка памяти возникает, когда мы забываем / не можем освободить выделенную память.

Dangling pointer

Глянем на другой пример:

```
int *p = (int *)malloc(100);
free(p); // now p is a dangling pointer

...
*p = 1;
```

Здесь мы освободили выделенную память, а потом воспользовались указателем на нее.

После `free` поинтер уже указывает на потенциально чужую память, которую нам трогать не стоит. Однако мы вполне себе можем, и в этом проблема.

Обычно такую проблему называют **dangling pointer** (**висячий указатель**).

Double free

И еще примерчик:

```
int *p = (int *)malloc(100);
free(p); // now p is a dangling pointer

...
free(p);
```

Я думаю, из названия все понятно. В данном случае мы либо случайно освобождаем чужую память, либо попадаем на undefined behavior.

А что делать то?

- Всегда проверяем на `NULL`! Многие из функций выше возвращают `NULL` при ошибке. Всегда стоит проверять на это, чтобы случайно не начать использовать null pointer. В т.ч. не стоит доверять пользователю, когда мы пишем свои функции (**особенно**, если мы пишем библиотеку) *Ну и бонусом сюда же: сначала освобождаем детей, а потом родителей!*)

```
int *p = (int *)malloc(100);
if (p == NULL) {
    perror("malloc failed");
    return -1;
}
```

```
int free_strings_array(char **arr, int arr_length) {
    if (arr == NULL) { // we don't trust the caller
        return -1;
    }

    // free children first
    for (int i = 0; i < arr_length; i++)
        free(arr[i]); // if arr == NULL then arr[i] is dereferencing NULL ptr
                       // which is undefined behavior!

    free(arr);
    return 0;
}
```

- = NULL после free. Как минимум поможет избежать негативных эффектов от double free, т.к. free(NULL) просто [ничего не делает](#).

```
int *p = (int *)malloc(100);

...
free(p);
p = NULL; // generally good practice

...
free(p); // does nothing
```

- Для каждой функций, которая возвращает указатель на данные, память под которые выделена malloc-ом (calloc-ом), имеет смысл писать еще одну функцию, которая будет эту память освобождать.

```
typedef struct {
    ...
} Image;

Image *read_image_from_file(char *filepath) {
    ...
    return ptr;
}

int free_image(Image *img) {
    ...
}
```

Хз, что тут еще придумать именно про malloc / calloc и free. В целом про memory safety в Cи можно чуть почитать [здесь](#).

А как обнаружить?

Что если наш код прекрасно работает, никаких segfault-ов не вылетает, но в нем все равно есть некоторые из проблем выше? Как их обнаружить? Например, те же утечки памяти не заметны сразу при исполнении.

На помощь приходят специальные инструменты. Есть два основных, которые помогут нам проверить программу на некорректное управление памятью: **AddressSanitizer** (+ **встроенный в него LeakSanitizer**) и **Valgrind memcheck**.

Основное отличие: *AddressSanitizer* - часть компилятора, когда *Valgrind* отдельная утилита.

Оба являются динамическими анализаторами и ищут ошибки во время исполнения программы (т.е. если во время исполнения некорректный код никогда не будет выполнен, то

оба инструмента ничего не скажут).

AddressSanitizer включается, как опция компилятора. *Valgrind* же запускается на уже скомпилированном исполняемом файле.

Оба умеют искать memory leaks, dangling pointers, да и в целом различные повреждения памяти.

Фрагментация

Четкого определения не будет, но фрагментация памяти это про ее неэффективное использование

Можно сказать, что это неэффективное использование свободного пространства в памяти компьютера

Внутренняя фрагментация

Выделение большего объёма, чем фактически требуется. Из-за этого вся избыточно выделенная память пропадает

Поступает запрос в ОС на выделение блока памяти, длиной N байт. Система неким образом (любым алгоритмом) выделяет кусок памяти.

В силу того, что алгоритмы выделения кусков памяти разные, часто реально выделяется не N байт, а N + K байт, где K - значение или 0, или вполне реальное.

Все «выделители» памяти работают таким образом. Обычно никогда не выделяется ровно столько памяти, сколько запрашивается процессом, т.е. внутри выделенного блока памяти есть неиспользованное пространство (K) — это есть **внутренняя фрагментация** – фрагментация внутри блока.

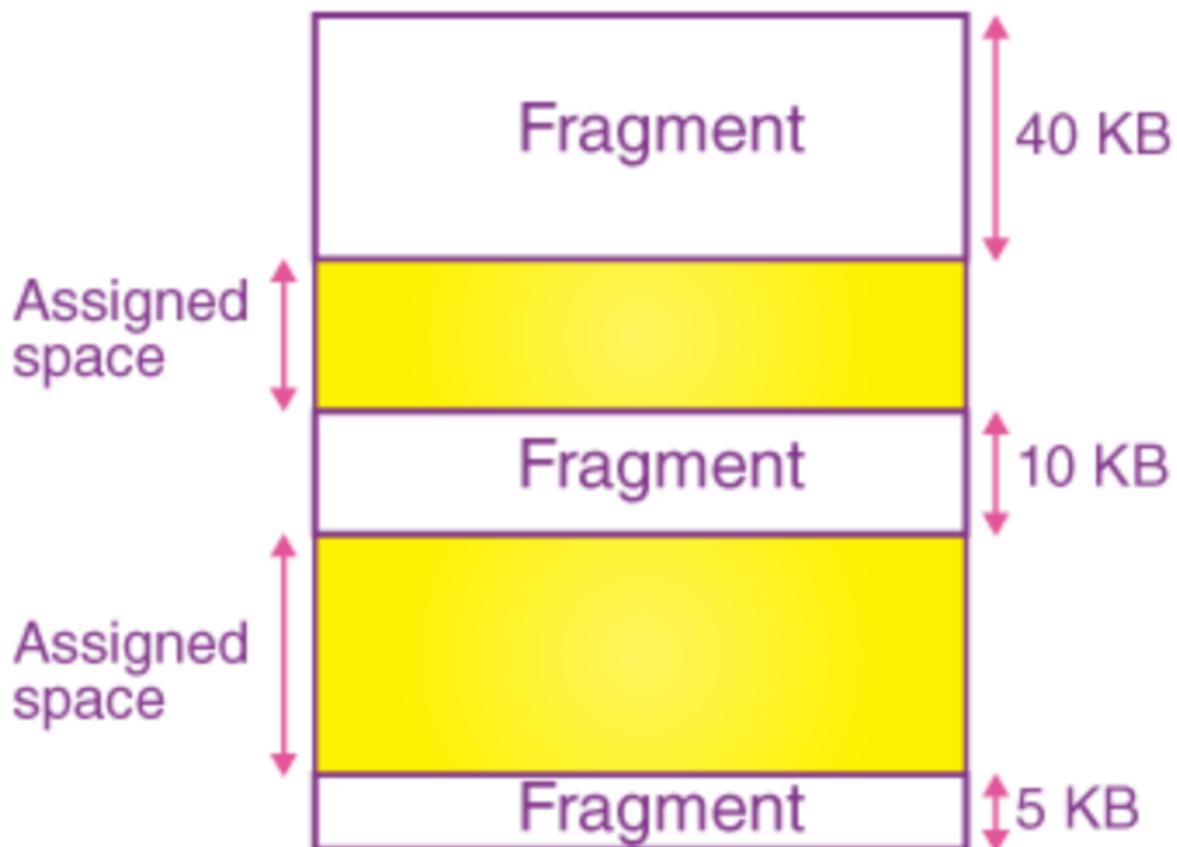
Эти K при использовании многих блоков накапливаются, они вроде бы и есть, но использовать их нельзя.

Внешняя фрагментация

Представим, что мы выделили несколько кусков памяти, а потом несколько из них освободили.

Поступает запрос на выделение большого куска памяти. Если суммировать освобожденные блоки памяти, то вполне хватит, но они разбросаны. Поэтому процессу память не выделится, будет получен отказ.

Возникла **внешняя фрагментация** – по отношению к блоку выделенной памяти она располагается снаружи.



External Fragmentation

source: <https://byjus.com/gate/external-fragmentation-in-os-notes/>

В примере слева мы не сможем выделить больше 40 кб, хотя в сумме у нас есть все 55 кб свободного пространства.

**(Язык С) Процедуры. Параметры.
Глобальные и локальные переменные.
Понятия области видимости, «времени
жизни», размещение в памяти.
Организация кадра. Соглашения о
вызовах (способы передачи аргументов)**

Основные понятия

Параметр - принятый функцией аргумент. Переменная в функции, которая будет содержать передаваемое снаружи входное значение.

Аргумент - это что **конкретно** и какой конкретной функции было передано. Входное значение **при вызове функции**.

Вызывающий код *передает аргумент в параметр*, который определен в члене спецификации функции.

Стоит отличать:

- формальный параметр - аргумент, указываемый при объявлении или определении функции (т.е. мы в объявлении прописываем что мы хотим получить и как)
- фактический параметр - аргумент, который мы реально (true) передаем

Подпрограмма или процедура ?

Часть с лекции [Парадигмы прогр-ния](#)

Подпрограмма -

1. проименованная или иным образом идентифицированная часть компьютерной программы, содержащая описание определённого набора действий ([с вики](#))
2. именованная часть программы, которую можно многократно вызывать для выполнения описанных в ней действий
3. именованная часть кода, предназначенная для повторного использования
4. выделенная переиспользуемая часть кода

Любое из этих определений, суть одна - переиспользуем часть кода (ее можно заименовать или выделить)

Подпрограмма представляет собой последовательность программных команд, которые выполняют конкретные задачи и объединены в единое целое. В различные языках программирования подпрограммы могут называться по-разному: **процедуры, функции**.

Подпрограммы подразделяются на функции, которые всегда возвращают результат, и **процедуры**, которые этого не делают. В С реализованы лишь функции, а процедуры получаются как функции типа void.

Процедура - обладает свойствами:

- переиспользуемое действие (подпрограмма)

- параметризация (формальные, фактические, по ссылке, по значению)
- то ли всегда возвращает результат, то ли нет (Паскаль и С)
- умеет не трогать чужое (собственные локальные переменные)
- рекурсия

Преимущества функций / подпрограмм этого:

- Разбиения прог-мы на простые шаги (база структурированного программирования и структур данных)
- Уменьшение дублированного кода
- Возможность повторного использования кода в других программах
- Разделение крупной программной задачи между различными программистами, или различными стадиями проекта
- Скрытие деталей реализации от пользователей подпрограммы (тип есть APIшка функции, а как она работает знать не надо)
- Улучшение прослеживания (большинство языков предоставляют способ получить след вызова, который включает в себя имена задействованных подпрограмм и, возможно, даже больше такой информации, как имена файлов и номера строк). Без декомпозиции кода на подпрограммы, отладка была бы серьезно затруднена.

Отступление

Сдавая экзамен, у проверяющего не возникло никаких вопросов, когда я сказал, что **процедура - функция, не возвращающая результата**

Область видимости объекта (переменной или функции) - то, в каких участках программы допустимо использование имени этого объекта.

Время жизни переменной может быть **глобальным и локальным**.

- Глобальное - в течение всего времени выполнения программы с ней ассоциирована ячейка памяти и значение.

Время жизни функции всегда глобально.

- Локальное - выделяется новая ячейка памяти при каждом входе в блок, в котором она определена или объявлена.

Время жизни объекта (или жизненный цикл) объекта - это время между созданием объекта и его уничтожением.

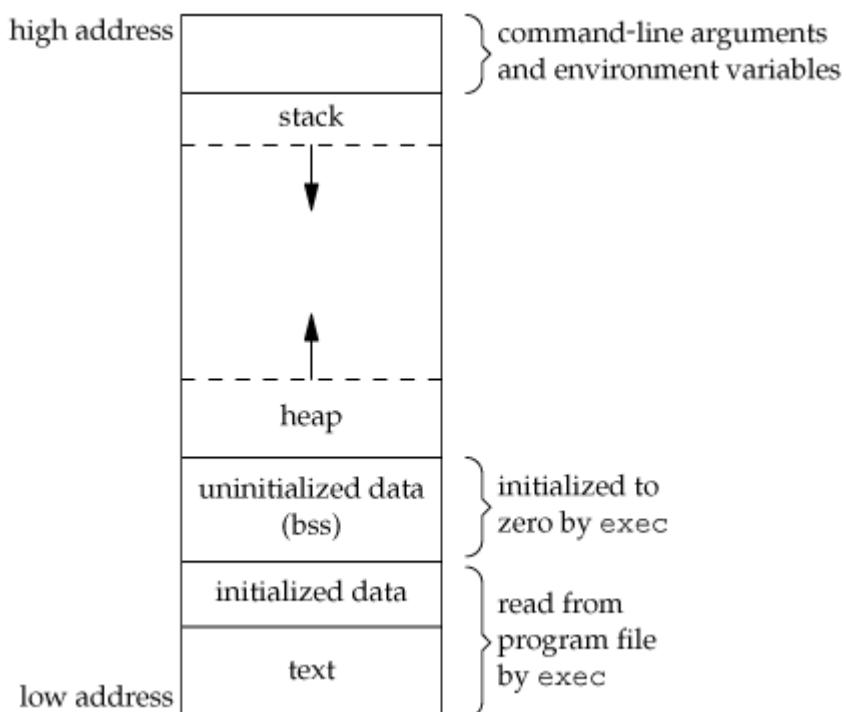
1. Переменная, объявленная глобально (т.е. вне всех блоков), существует на протяжении всего времени выполнения программы.
2. Локальные переменные (т.е. объявленные внутри блока) с классом памяти register или

`auto`, имеют время жизни только на период выполнения того блока, в котором они объявлены. Если локальная переменная объявлена с классом памяти `static` или `extern`, то она имеет время жизни на период выполнения всей программы.

`register` - переменная *возможно* разместится в регистрах; `auto` - ставит переменной локальную область видимости, в С используется по умолчанию, `defines a local variable as having a local lifetime`. Сохраняется значение внутри 1 вызова `static` - переменная будет сохранять свое значение между вызовами функций `extern` - переменная, объявленная в другой области того же файла или другого файла, может быть доступна из любого места программы.

Организация кадра

Перед началом работы функция должна захватить в стеке область памяти под свои локальные переменные.



Разделы памяти для запущенного процесса

- Пояснение к картинке

- Аргументы командной строки и переменные среды: в этом разделе хранятся аргументы, передаваемые программе перед запуском, и переменные среды.

Интересующий(ся) читатель, в качестве дз, поищет информацию про переменные среды

- Стек: хранятся все параметры функции, адреса возврата и локальные переменные функции. Структура вида LIFO (last in first out). Растет вниз в памяти (от больших

адресов к меньшим) по мере выполнения новых вызовов функций. Позже мы рассмотрим стек более подробно.

- Куча: динамически выделяемая память (malloc). Куча растет вверх в памяти (от более низких адресов памяти к более высоким), поскольку требуется все больше и больше памяти.
- Неинициализированные данные (сегмент Bss): здесь хранятся все неинициализированные данные. Он состоит из всех глобальных и статических переменных, которые не инициализируются программистом. Ядро инициализирует их арифметическим 0 по умолчанию.
- Инициализированные данные (сегмент данных): здесь хранятся все инициализированные данные. Он состоит из всех глобальных и статических переменных, которые инициализируются программистом.
- Текст: это раздел, в котором хранится исполняемый код. Отсюда загрузчик загружает инструкции и выполняет их. Часто только для чтения.

Некоторые из основных регистров:

- %eip: Регистр указателя инструкций. Хранит адрес следующей инструкции для выполнения. После выполнения каждой инструкции ее значение увеличивается в зависимости от размера инструкции.
- %esp: Регистр указателя стека. Хранит адрес вершины стека - последнего элемента в стеке. Стек растет вниз в памяти, таким образом, %esp указывает на значение в стеке по самому низкому адресу памяти.
- %ebp: Регистр базового указателя. Обычно устанавливается в %esp в начале функции. Это делается для сохранения вкладки параметров функции и локальных переменных. Доступ к локальным переменным осуществляется путем вычитания смещения из %ebp, а доступ к параметрам функции осуществляется путем добавления к нему смещения, как вы увидите в следующем разделе.

Примерчик из жизни

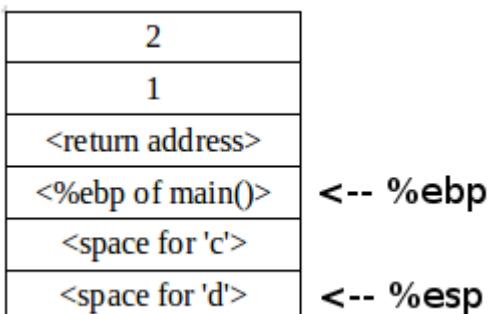
Рассмотрим код:

```
void func(int a, int b)
{
    int c;
    int d;
    // some code
}
void main()
{
    func(1, 2);
    // next instruction
}
```

Предположим, что наш %eip указывает на вызов func в main. Будут предприняты

следующие шаги:

1. Обнаружен вызов функции, помещаем параметры в стек справа налево (в обратном порядке). Таким образом, сначала будет нажато 2, а затем 1.
2. Нам нужно знать, куда возвращаться после завершения func, поэтому поместим адрес следующей инструкции в стек.
3. Найдем адрес func и установим %eip на это значение. Управление передано функции func().
4. Поскольку мы находимся в новой функции, нам нужно обновить %ebp. Перед обновлением мы сохраняем его в стеке, чтобы потом вернуться обратно в main. Таким образом, %ebp помещается в стек.
5. Устанавливаем %ebp равным %esp. %ebp теперь указывает на текущий указатель стека.
6. Вставляем локальные переменные в стек/резервное место для них в стеке. %esp будет изменен на этом шаге.
7. После завершения func нам нужно сбросить предыдущий кадр стека. Поэтому установим %esp обратно в %ebp. Затем извлечем предыдущий %ebp из стека и сохраните его обратно в %ebp. Таким образом, регистр базового указателя указывает туда, куда он указывал в main.



Собственно, так это и будет выглядеть

Итого, как все кладется в стек при вызове функций:

- Аргументы в обратном порядке
- Адрес возврата (чтобы знали куда прыгать по выходе)
- ebp функции вызова (чтобы могли по выходе юзать её локальные переменные)
- Локал переменные функции вызова

Соглашение о вызове(calling convention)

- описание особенностей вызова подпрограмм

Определяет собой:

- способы передачи параметров подпрограмм;
- способы вызова (передачи управления) подпрограмм;

- способы передачи результатов вычислений, выполненных подпрограммами, в точку вызова;
- способы возврата (передачи управления) из подпрограмм в точку вызова.

Какие вообще существуют варианты реализации тех или иных пунктов выше можно глянуть [тут](#), более конкретные соглашения далее

Некоторые примеры

- *cdecl* - то, что рассмотрено выше. Аргументы передаются через стек, справа налево. Если размер аргумента меньше 4х байт - размер расширяется. Очистку стека производит вызывающая программа.
- *pascal* - соглашение о вызовах, используемое компиляторами для языка Паскаль. Также применялось в ОС Windows 3.x. Аргументы процедур и функций передаются через стек, слева направо. Указатель на вершину стека (значение регистра *esp*) на исходную позицию возвращает вызываемая подпрограмма. Изменяемые параметры передаются только по ссылке. Возвращаемое значение передаётся через изменяемый параметр *Result*. Параметр *Result* создаётся неявно и является первым аргументом функции.
- *stdcall* или *winap* - соглашение о вызовах, применяемое в ОС Windows для вызова функций *WinAPI*. Аргументы функций передаются через стек, справа налево. Очистку стека производит вызываемая подпрограмма.
- *thiscall* - соглашение о вызовах, используемое компиляторами для языка C++ при вызове методов классов в объектно-ориентированном программировании. Аргументы функции передаются через стек, справа налево. Очистку стека производят вызываемая функция. Соглашение *thiscall* отличается от *stdcall* соглашения только тем, что указатель на объект, для которого вызывается метод (указатель *this*), записывается в регистр *esx*

Чуть больше описано [тут](#), интересное замечание про *libffi* (библиотека для вызовы функций с использованием разных соглашений)

Почему нет одного универсального соглашения ? + использование

1. Соглашение о вызове выбирается во время оптимизации для увеличения скорости выполнения программы или для уменьшения её размера (уменьшения числа инструкций).

Т.е. каждое соглашение будет оптимальным в тех или иных условиях, а можно и [изменять](#) те или иные соглашения

2. При вызове функций из системных или сторонних библиотек необходимо применять соглашения о вызовах, выбранное на этапе сборки этих библиотек.
3. При анализе машинного кода с целью получения текста программы на языке высокого уровня генерированные компилятором типовые прологи и эпилоги позволяют распознать адреса начал и концов функций.

Выделение памяти: стековый аллокатор, implicit free list

Stack Allocator

Аллокатор (англ. *Allocator*) — или **распределитель памяти** в программировании — специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Стековый аллокатор - это умная штука, которая позволяет управлять памятью, как стеком.

Предисловие

- Мы уже имели дело с аллокаторами памяти
- Они нужны, чтобы работать со структурами данных динамического размера
- Аллокатор работает с кучей (heap)
- Для аллокатора куча - множество блоков разного размера
- Блок - непрерывный кусок виртуальной памяти, который может быть либо выделенным (allocated), либо свободным (free).

Требования к аллокатору

- Последовательность запросов malloc и free - произвольная
 - Нельзя полагаться на порядок запросов
 - Но мы предполагаем, что free вызывается на участке, который был выделен
- Немедленный ответ на запрос
 - Нельзя буферизировать запросы или переупорядочивать
- Используется только куча
 - Все нескалярные структуры данных, которыми пользуется аллокатор, должны лежать в куче
- Выравнивание блоков
 - Нужно, чтобы в блоке могли размещаться данные любого типа
 - В большинстве систем выравнивается по 8 байт
- Нельзя модифицировать выделенные блоки
 - Можно манипулировать только свободными блоками

Цели

- Максимизация пропускной способности (throughput)
 - Количество запросов, выполняющихся в единицу времени

- Нужно уменьшать среднее время на запрос к аллокатору
- Максимальная утилизация памяти (memory utilization)
 - Полезная нагрузка (payload) - сколько памяти действительно было запрошено
 - Нам нужно максимизировать суммированную полезную нагрузку для всех запросов относительно размера кучи

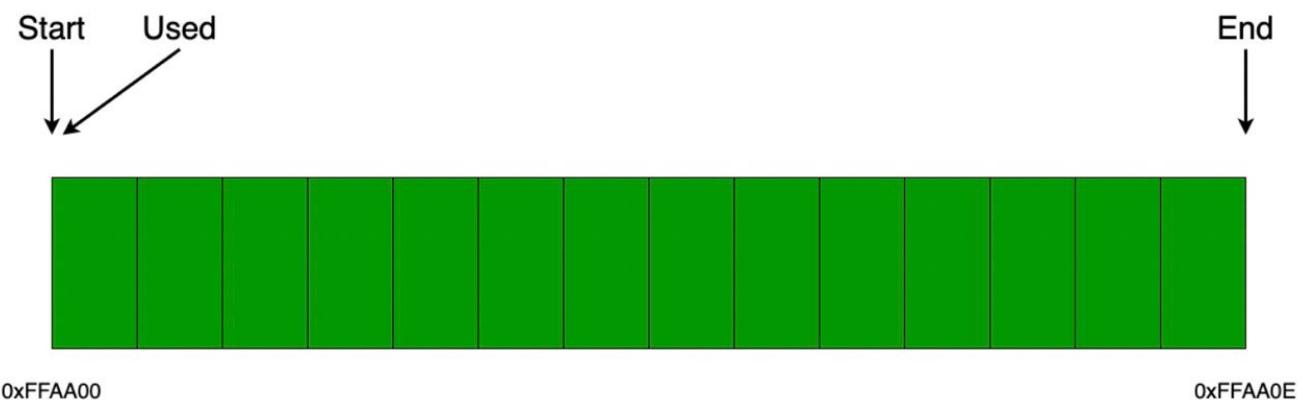
Мы хотим быстро и без фрагментации

Самый простой аллокатор: Linear Allocator

- Куча - массив с указателем p на начало
- malloc(size): увеличить указатель p на size, вернуть новый указатель
- free(ptr): просто return, ничего не делать

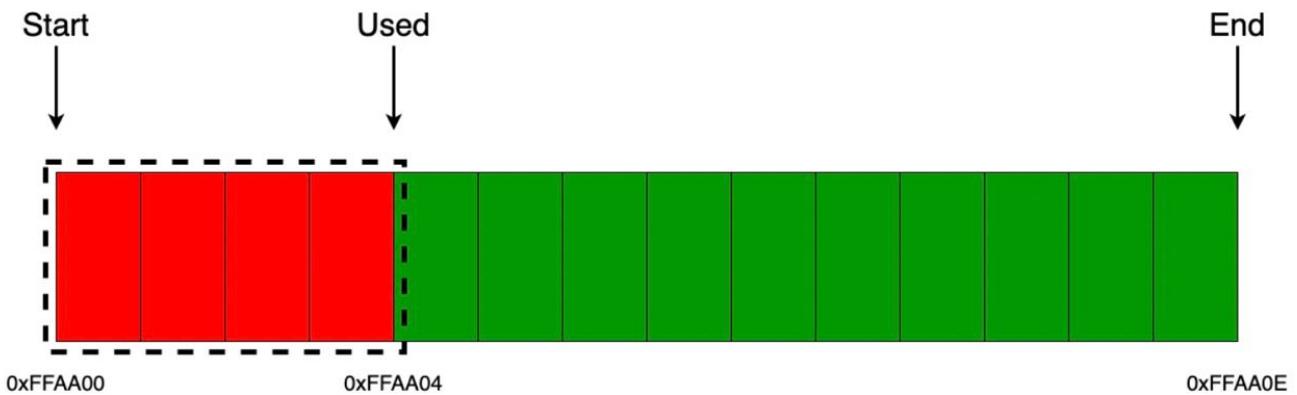
Что мы получили:

- Хорошая пропускная способность, все запросы за константу
- Отвратительная утилизация памяти - не переиспользуем свободные блоки. Происходит это из-за того, что мы не знаем размеры блоков, которые выделяли.



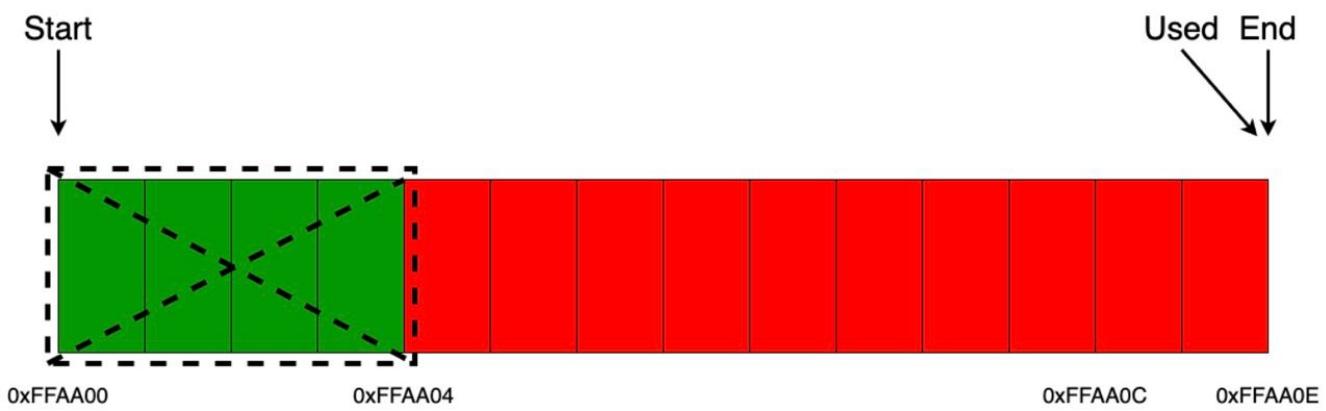
Представим, что в аллокатор поступил запрос на выделение 4 байт памяти. Действия аллокатора на исполнения этого запроса будут следующие:

- проверить достаточно ли памяти для выделения;
- сохранить текущий указатель used, который в дальнейшем будет отдан пользователю, как указатель на блок выделенной памяти из аллокатора;
- сместить указатель used на величину равную объему выделенного блока памяти, т.е. на 4 байта.



Дальше, например, приходит запрос на выделение 8 байт и, соответственно, действия аллокатора будут точно такими же вне зависимости от размера выделяемого блока памяти. Выделяем так до конца

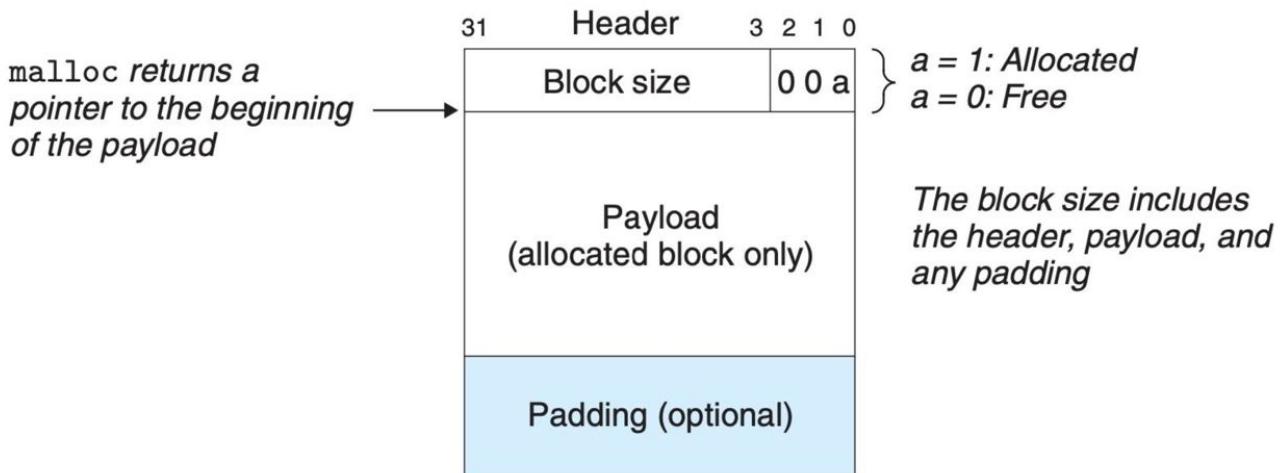
Отлично, теперь самое время поговорить об освобождении памяти. Как уже отмечалось ранее, данный вид аллокаторов не поддерживает выборочное освобождение определенных блоков памяти. То есть, если провести тонкую аналогию с *malloc/free*, имея указатель, скажем, на **0xFFAA00**, мы могли бы освободить этот блок памяти



но вот линейный аллокатор нам этого позволить не может. Все, что мы можем сделать, это освободить всю занятую память целиком внутри аллокатора и продолжить работать с ним, как с совершенно пустым.

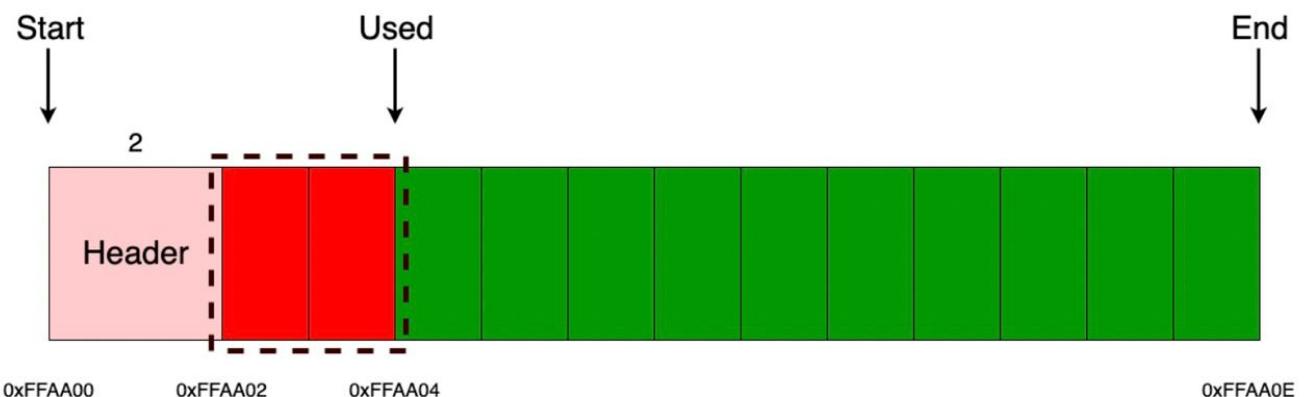
Stack allocator

- Как различать границы блоков и понимать, свободен блок или нет?
- Будем хранить все нужное в самом блоке

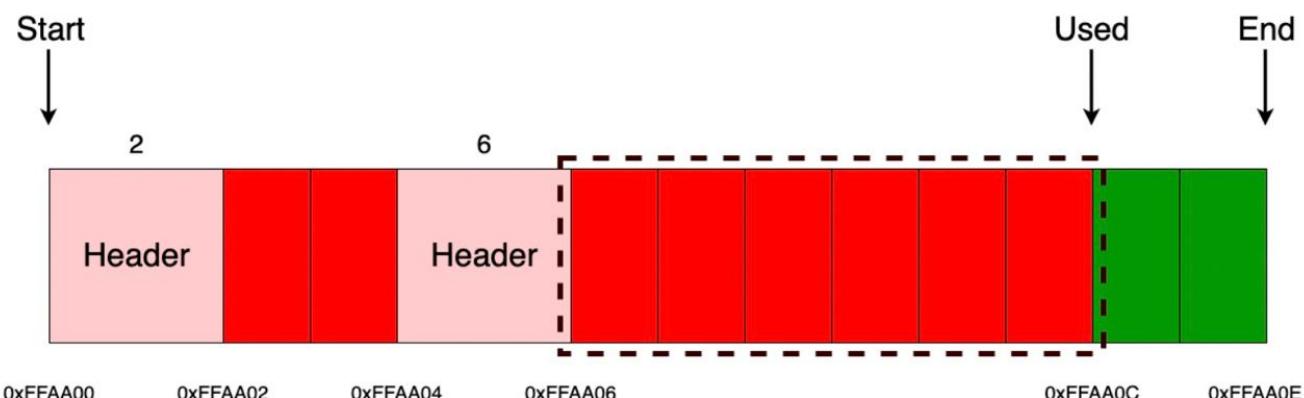


Header содержит информацию о размере блока и о том, занят он или нет.

Когда приходит запрос на выделение памяти, помимо выделения некоего ее объема памяти, запрашиваемого пользователем, мы еще дополнительно выделяем заголовок (пользователь с ним никак не будет взаимодействовать), в котором храним сведения о том, сколько было выделено байт (в данном примере размер заголовка составляет 2 байта). Например, если пришел запрос на выделение 2 байт, то состояние аллокатора будет точно таким же, как на рисунке ниже. Важно отметить то, что пользователю будет отдан указатель не на заголовок, а на блок, следующий сразу за заголовком, то есть в данном примере это блок с адресом **0xFFAA02**.

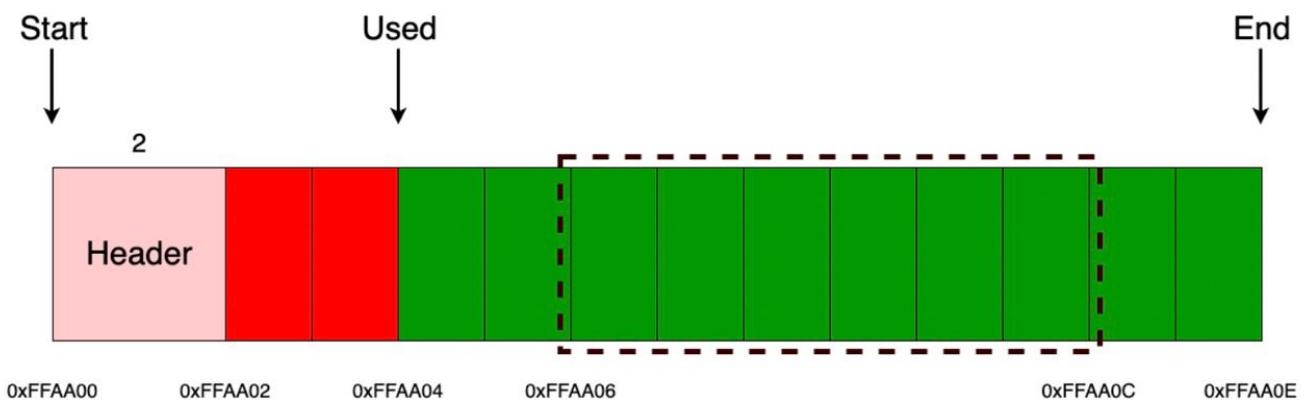


Аналогичная ситуация будет и, например с выделением 6 байт.



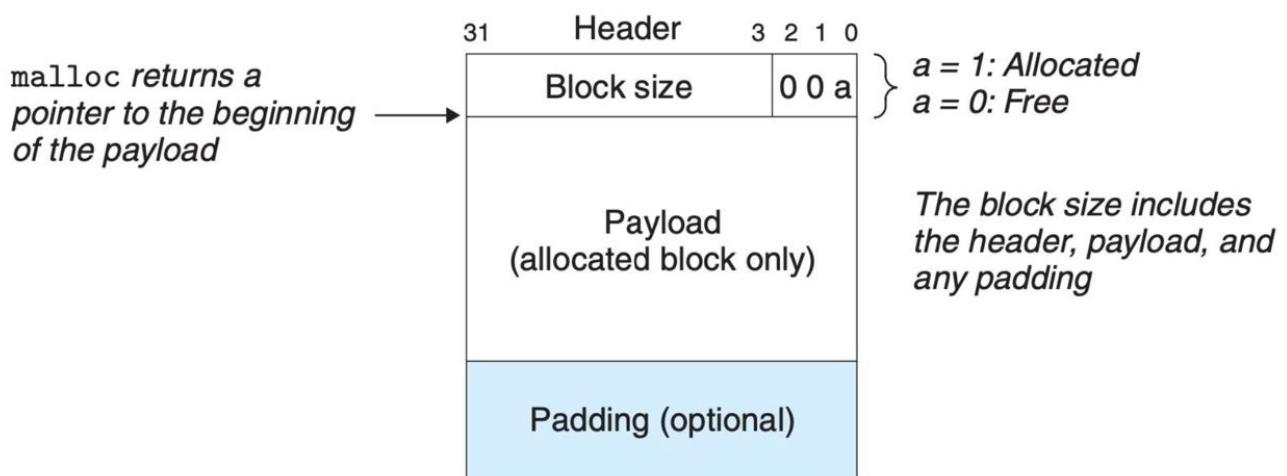
А вот с освобождением все немного поинтереснее. Для начала от указателя, который

пользователь просит освободить, нужно отнять размер заголовка, после чего разыменовать значение и уже только после этого сдвинуть указатель *used* на размер заголовка вместе с размером блока, полученного из заголовка.



Implicit free list

Давайте пока что использовать такую же структуру блока как в stack allocator



Размещение выделенного блока(placement)

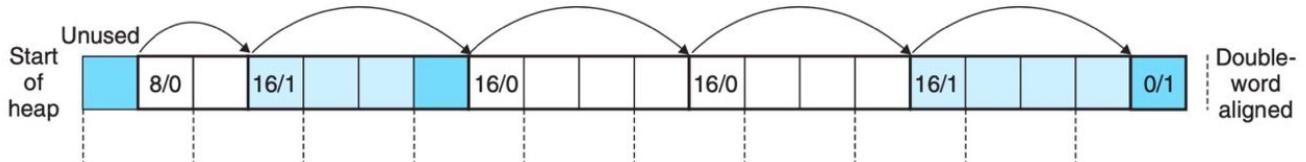
При запросе аллокации - поиск подходящего свободного блока. А как искать? Есть разные **политики размещения**!

- **first fit** - берем первый попавшийся подходящего размера
 - Обычно свободные блоки побольше оказываются в конце списка
 - В начале свободные блоки меньше ⇒ поиск блока побольше займет больше времени
- **next fit** - начинаем поиск там, где закончился предыдущий
 - Есть вероятность, что следующий подходящий блок - остаток предыдущего
 - Может работать быстрее, чем first fit
 - Хуже утилизирует память
- **best fit** - перебираем все блоки и ищем подходящий с наименьшим размером

- Лучше утилизирует память
- Хуже по времени - бежим по всей куче

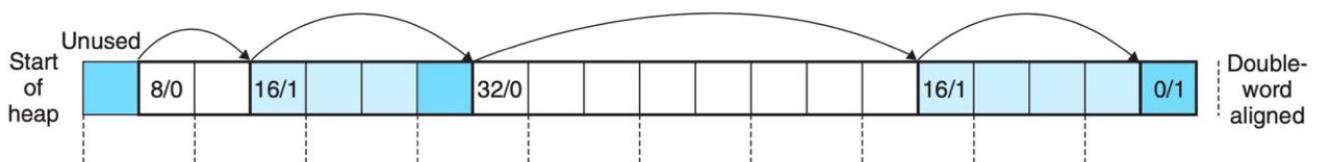
Объединение свободных блоков (coalescing)

При освобождении свободные блоки могут оказаться рядом.



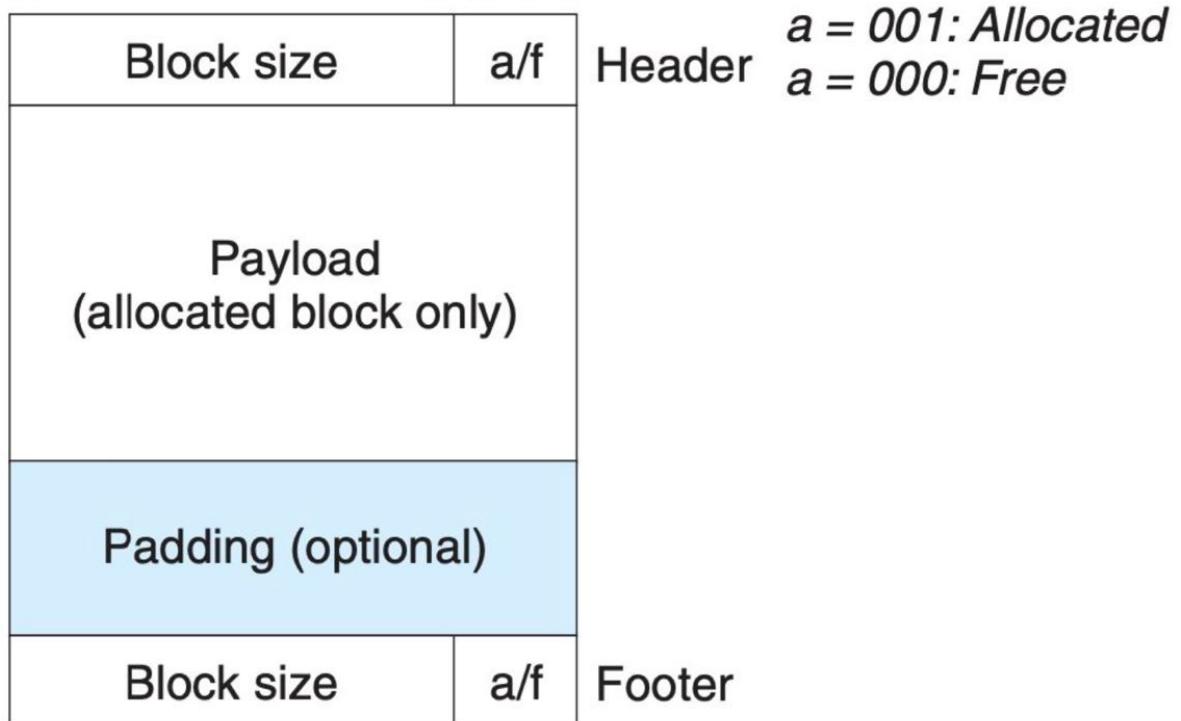
- Получили ложную фрагментацию:
 - Запрос на 4 слова не выполнится, хотя место есть
 - У нас два блока с payload = 3 слова
- Блоки надо объединять

Если мы будем использовать такую же структуру блока, как в stack allocator, то мы будем долго объединять свободные блоки(вернее за константу со следующим блоком и за линию с предыдущим). Да и вообще любую операцию будем выполнять за $O(\text{кол-во блоков})$. Быстро перемещаться мы пока что можем только слева направо, но не наоборот.



- Последовательность занятых и свободных блоков
 - Односвязный список свободных блоков
 - В конце - маркер окончания списка
- Время на операцию - линейное от количества всех блоков
 - Это недостаток, с которым в будущем будем бороться
- Выравнивание
 - Появляется минимальный размер блока - 2 слова

Для этого воспользуемся методом граничных маркеров и получим новую структуру блока:



Теперь мы можем быстро перемещаться между блоками в обе стороны(вся информация о размерах в *header* и *footer*).

На картинке ниже показано, как будет происходить объединение при освобождении блока размером n

m1	a
m1	a
n	a
n	
n	a
m2	a
m2	a

m1	a
m1	a
n	f
n	
n	f
m2	a
m2	a

Case 1

m1	a
m1	a
n	a
n	
n	a
m2	f
m2	f

Case 2

m1	a
m1	a
$n+m2$	f
$n+m2$	
$n+m2$	f

m1	f
m1	f
n	a
n	
n	a
m2	a
m2	a

$n+m1$	f
$n+m1$	f
m2	a
$m2$	
$m2$	a

Case 3

m1	f
m1	f
n	a
n	
n	a
m2	f
m2	f

Case 4

$n+m1+m2$	f
$n+m1+m2$	f

- Получили константное время в каждом случае
- Подход легко обобщить на разные типы аллокаторов
- Тратим много памяти на header и footer
- Можем оптимизировать:
 - Можем избавиться от футера у выделенных блоков

Implicit Free Lists - что получили?

- Линейное время от количества всех блоков на аллокацию
- Константное время на освобождение
- Очень просто реализовать
- Редко используется из-за скорости malloc, но при этом довольно в определенных случаях может подойти
- Разделение и объединение может распространяться почти на все аллокаторы!

Реализацию Implicit Free Lists на C можно посмотреть в презентации

[Dynamic Memory Allocation.pdf](#)

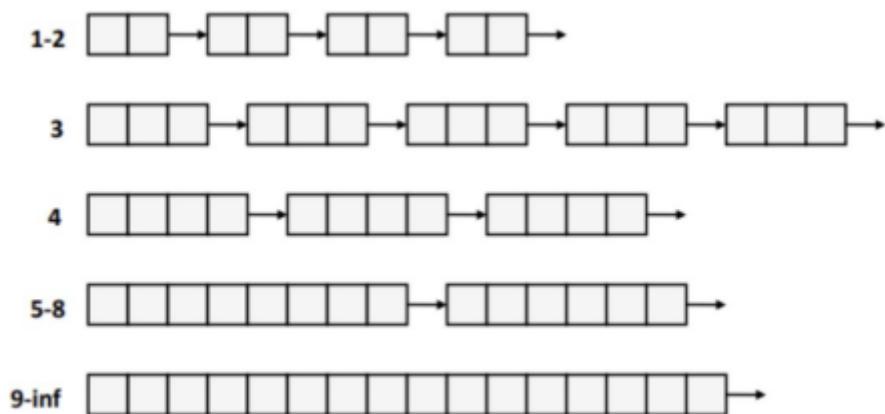
Для полного понимания темы рекомендую побежаться по презентации.

Выделение памяти: метод двоичных близнецов, SLAB-аллокатор.

Аллокатор (англ. *Allocator*) или распределитель памяти в программировании — специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Предисловие (про segregated free lists, segregated fits)

- Делаем массив списков со свободными блоками определенных размеров
- Множество блоков разбиваем на классы по размерам
- Политик того, как разбить блоки на классы - много:
 - Можно по степеням двойки (эта идея выстреливает в близнецах)
 - Можно маленькие размеры выделять в отдельные классы



Аллокация и освобождение

sbrk() - функция выделения памяти для кучи

- Чтобы выделить блок размера n
 - Ищем в списке свободных блоков подходящего класса
 - Если нашли свободный блок - выделяем
 - Разделить, а остаток поместить в нужный список - optional
 - Если не нашли, пробуем искать в списке блоков большего размера
- Перебрали все списки и не нашли - что делать?
 - Просим больше памяти у ОС при помощи sbrk()
 - Выделяем из новой памяти блок нужного размера
 - Остаток выделяем в отдельный блок и вставляем в нужный список
- Освобождение блока
 - Вставка блока в нужный список
 - Можем объединять блоки со вставкой в нужный список - optional

В итоге лучше используем память (проход с first fit почти меняется на best fit)

Лучше используется время (исследуем не всю кучу, а какую-то часть)

Segregated Fits

- Каждый список - явный или неявный (как описывалось ранее)
 - В списке - блоки разных размеров!
- Выделение блока:
 - Бежим по нужному списку по политике first fit
 - Нашли - делим, остаток отправляем в нужный список
 - Не нашли - ищем в списке класса больших размеров
 - Перебрали все списки? Просим памяти у ОС, выделяем нужный блок, остаток - помещаем в нужный список
- Освобождение блока:

- Объединяем блоки и результат отправляем в нужный список

Плюс в целом желательно, чтобы вы ознакомились с понятием аллокатора и какими-то основами этой темы (первые слайдов 10-15 из презентации с последней практики)

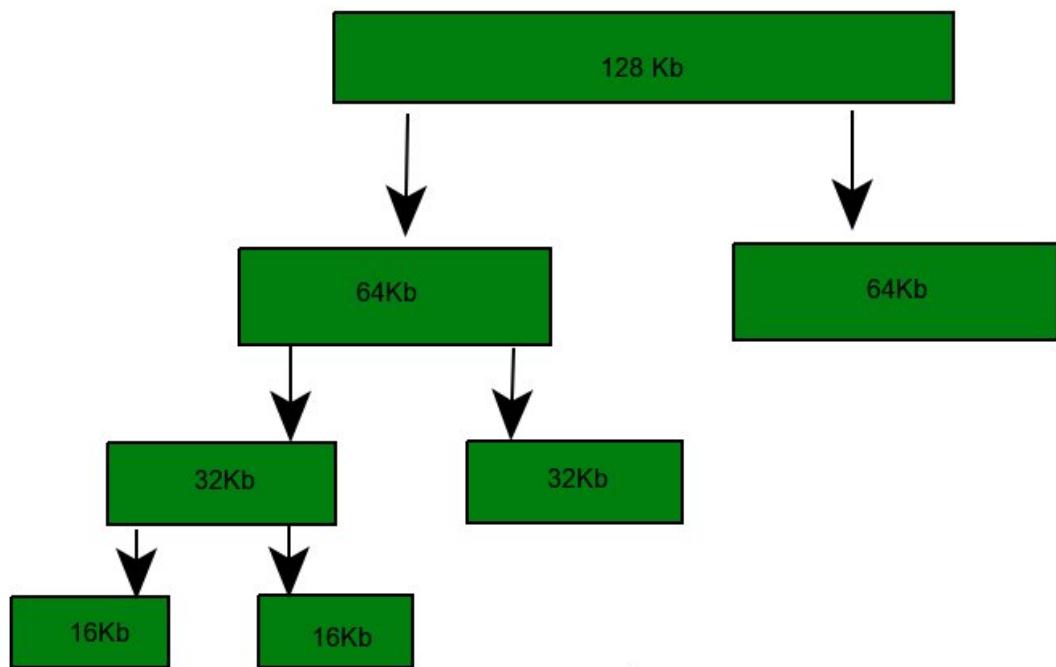
Метод близнецов

Одна из версий алгоритма распределения друзей была подробно описана Дональдом Кнутом в первом томе книги «Искусство компьютерного программирования».

Используется в составе многих современных операционных систем для динамического распределения памяти в ядре системы, драйверах или в других ответственных компонентах системы, критичных к скорости работы.

Ядро линукса основано на методе двоичных близнецов (с модификациями, конечно) !

- Основная идея - segregated fits, **каждый класс - степень двойки**
- Всего в куче 2^m слов
- Держим список для блоков размером 2^k , $0 \leq k \leq m$
- Изначально у нас один блок размером 2^m
- Большая и понятная картинка



Виды близнецов (доп нагрузка)

Основные реализации этого алгоритма таковы:

- Бинарная система близнецов (как в примере и на пикчах, и в целом вопрос о них)
- Система близнецов Фибоначчи (когда выделяем по 1,1,2,3,5 и тд), [тут-с](#) пример

Пример

Это часть примера с вики, полностью смотреть [тут](#)

- Хотим выделять блоки размером 2^k
- Надо найти первый доступный блок размером 2^j , $k \leq j \leq m$
- Рекурсивно:
 - Если $j = k$ - все закончилось !
 - Если нет - рекурсивно делим блок пополам, пока j не станет равным k .
 - При делении оставшуюся часть (близнеца) отправляем в нужный список

Вот как происходит выделение блока 34К при минимальном размере 64К.

(2^0 на пикче - не степень, а порядок блока)

Step	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K	64 K
1	2^4															
2.1	2^3															
2.2	2^2				2^2											
2.3	2^1		2^1		2^2											
2.4	2^0	2^0	2^1		2^2											
2.5	A: 2^0	2^0	2^1		2^2											

source: https://en.wikipedia.org/wiki/Buddy_memory_allocation

Вот освобождение блока D (два мини блока 2^1 собираются в один большой 2^2)

6	A: 2^0	D: 2^0	2^1	D: 2^1	2^1	2^3
7.1	A: 2^0	C: 2^0	2^1	2^1	2^1	2^3
7.2	A: 2^0	C: 2^0	2^1	2^2		2^3

source: https://en.wikipedia.org/wiki/Buddy_memory_allocation

на красный цвет не обращайте внимания, он используется в расширенном примере в вики

- Еще один примерчик

Рассмотрим метод близнецов на простом примере, в котором алгоритм применяется для обработки 1024-байтового блока с минимальным размером участков в 32 байта. Распределитель использует битовую карту для отслеживания каждой 32-битовой порции блока: если бит установлен, то соответствующий участок занят. Она также поддерживает список свободных буферов любого допустимого размера (по степеням двойки в диапазоне от 32 до 512). Изначально блок представляет собой единый буфер. Представим, что произойдет с ним при поступлении некоей последовательности запросов на выделение памяти и ответную реакцию распределителя.

1. allocate(256). Блок делится на два близнеца, А и А'; блок А' поступает в список свободных 512-байтowych буферов. Затем буфер А разбивается на В и В'. В' заносится в список свободных 256-байтных буферов, а буфер В передается клиенту.
2. allocate(128). Распределитель обнаруживает, что список свободных 128-байтowych буферов пуст. Тогда он проверяет список 256-байтowych буферов, изымает оттуда В' и разделяет его на С и С'. После этого буфер С' помещается в список свободных 128-байтowych буферов, а буфер В возвращается клиенту.
3. allocate(64). Распределитель выясняет, что список свободных 64-байтowych буферов пуст. Тогда он обходит список 128-байтowych буферов, удаляет оттуда С' и дробит его на D и D'. Последний передается в список свободных 64-байтowych буферов, а буфер D возвращается клиенту.
4. allocate(128). Распределитель узнает, что списки свободных 128- и 256- байтowych буферов пусты. Он проверяет список свободных 512-байтowych буферов и удаляет оттуда буфер А'. Затем А' разделяется на два близнеца Е и Е', а буфер Е — на F и F'. После этого буфер Е' переносится в список свободных буферов размером 256 байтов, а буфер F — в список 128-байтowych буферов. Буфер F передается клиенту.

А в чем минусы ?

- Страдаем от внутренней фрагментации (особенно когда размеры блоков не близки к степеням двойки). Это решается SLAB-аллокацией

- Если сначала взять много маленьких блоков и освободить каждого второго (то есть ровно одного из близнецов) тогда не будет происходить слияние → не сможем запросить половину памяти

А плюсы ?

- Быстрый поиск и объединение
 - Все блоки близнецов различаются единственным битом



- Вроде first-fit, но близки к best-fit
- Как правило, метод близнецов реализуется с использованием двоичного дерева для представления используемых или неиспользуемых разделенных блоков памяти.

SLAB-allocator

Slab - плита

- Что может помочь в понимании этой темы ? Disclaimer.

Чтобы лучше понять это, стоит рассматривать аллокатор как целостную систему управления памятью в рамках ОС

- Чтобы решить проблему внутренней фрагментации, в системе Linux есть второй механизм выделения памяти — распределитель фрагментов (slab allocator), выбирающий блоки памяти при помощи «приятельского» алгоритма, а затем нарезающий из этих блоков более мелкие куски и управляющий ими по отдельности.

Э. Таненбаум Современные операционные системы

Этот метод был впервые введен в **SunOS** Джеком Бонвиком и сейчас широко используется в ядрах многих операционных системах Unix, включая **FreeBSD** и **Linux**.

Системы, использующие slab:

1. [AmigaOS](#) (введено в 4.0)
2. [DragonFly BSD](#) (введено в релизе 1.0)
3. [FreeBSD](#) (введено в 5.0)

4. [Haiku](#) (введено в alpha 2)
5. [HP-UX](#) (введено в 11i)
6. [Linux](#) (введено в ядре 2.2)
7. [NetBSD](#) (введено в 4.0)
8. [Solaris](#) (введено в 2.4)

Сама суть

- Нейросеть

Плюсы SLAB-аллокации:

1. Эффективное использование памяти: SLAB-аллокатор предварительно выделяет блоки памяти фиксированного размера, что позволяет эффективно использовать доступную память без фрагментации.
2. Улучшенная производительность: Благодаря предварительному выделению памяти и повторному использованию освобожденных блоков, SLAB-аллокация может снизить накладные расходы на выделение и освобождение памяти, что приводит к улучшенной производительности системы.

Минусы SLAB-аллокации:

3. Потребление памяти: Предварительное выделение блоков памяти может привести к избыточному потреблению памяти, особенно если размеры объектов варьируются значительно или если система не использует все выделенные блоки.
4. Ограничение размера объектов: SLAB-аллокация предполагает использование блоков фиксированного размера, поэтому она может быть неэффективной для объектов, размер которых превышает размер блока.
5. Сложность реализации: Реализация SLAB-аллокатора может быть сложной и требовать дополнительных усилий по сравнению с другими методами управления памятью.

- Нейросеть 4.0

SLAB-аллокатор — это подсистема управления памятью в операционной системе, которая ускоряет работу ядра путем кэширования объектов данных. Особенно эффективна она при работе с данными фиксированного размера.

SLAB-аллокатор был впервые реализован в ядре операционной системы Solaris 2.4 в 1994 году, впоследствии был портирован на другие ОС, в том числе и на Linux.

Основная идея SLAB-аллокатора заключается в том, чтобы разделить разные типы объектов по разным кэшам. Внутри каждого кэша объекты делятся на группы, которые называются SLAB-ами. SLAB — это набор объектов одного типа, которые размещены на непрерывных страницах физической памяти. При запуске или завершении работы какого-либо процесса системе необходимо выделить или освободить память под большое количество объектов. Если для всех объектов память будет выделяться или

освобождаться индивидуально, то это может занять значительное время. Благодаря SLAB-аллокатору система может выделить или освободить память под все объекты SLAB-а сразу.

SLAB-аллокатор существенно снижает фрагментацию памяти, уменьшает время, затрачиваемое на выделение и освобождение памяти, а также улучшает производительность системы за счет поддержания **горячих** данных в кэше процессора.

Система SLAB-аллокатора включает в себя три основных компонента:

1. Объекты: Это основные данные, с которыми работает аллокатор. Они имеют определенный размер, который обычно заранее определен.
2. SLABы: SLAB является коллекцией объектов одинакового размера. SLAB может находиться в одном из трех состояний – полностью пуст, частично заполнен или полностью заполнен.
3. Кэши: Кэши содержат SLABы разного состояния. Каждый кэш занимается управлением SLABами с объектами определенного размера или типа. Кэш создается для каждого уникального типа объекта.

Основная идея SLAB-аллокатора заключается в том, что когда потребуется новый объект, он будет взят из слаба, который уже содержит объекты, иначе будет создан новый слаб. Когда объект больше не нужен, он возвращается в свой слаб, а не освобождается. Это позволяет быстро выделять и освобождать память под объекты одного размера, а также сокращает фрагментацию памяти и поддерживает **горячую** память.

SLAB-аллокатор предотвращает неэффективное использование памяти за счет того, что объединяет свободные объекты вместе и минимизирует необходимость взаимодействия с системой управления памятью ядра, тем самым повышая производительность и предотвращая фрагментацию памяти.

Фундаментальная идея способа основывается на результатах наблюдения, показывающих, что некоторые объекты данных ядра часто создаются и уничтожаются после того, как перестают быть нужными. Таким образом, при каждом выделении памяти для объектов такого типа затрачивается некоторое время для нахождения наиболее подходящего места для этого объекта. Кроме того, освобождение памяти после уничтожения объекта способствует большей фрагментации памяти, которая создает дополнительную нагрузку на ядро для реорганизации памяти.

В случае же с распределением slab, при использовании определённых системных вызовов, участки памяти, подходящие для размещения объектов данных определённого типа и размера, заранее предопределены. Если поступает запрос на выделение памяти для объекта данных определённого размера, он может мгновенно удовлетворить запрос уже выделенным слотом. Уничтожение объектов не освобождает память, а только открывает слот, который помещается в список свободных слотов распределителем slab. Следующий вызов для выделения памяти того же размера вернет слот памяти, не используемый в данный момент. Этот процесс устраняет необходимость в поиске подходящего участка памяти и значительно снижает фрагментацию памяти.

Когда происходит очищение объекта, он на самом деле не освобождается, а резервируется как кеш, который может использоваться непосредственно при распределении следующего запроса.

- Терминология **кэша** и **slab**

- **Кэш:** кэш представляет собой небольшой объём очень быстрой памяти. Здесь мы используем кэш как память для хранения разных объектов (дескриптор процесса, каждая группа объектов +- похожа по свойствам) Каждый кэш способен хранить только один тип объектов.
 - **объекты одного типа имеют одинаковый размер**
 - поэтому объекты одного вида могут быть объединены в цепочки, в которых места для новых объектов могут быть заранее предопределены

Дескриптор процесса - системные данные, используемые ядром в течении времени **жизни процесса**. Дескриптор процесса резервируется ядром при образовании процесса и освобождается при его завершении.

- **Slab** представляет собой непрерывный участок памяти, обычно составленный из нескольких физических смежных страниц. Кэш **состоит из одного или более slab'ов**.

В этом контексте slab — это одна или более смежных страниц в памяти, содержащих заранее выделенные участки памяти.

Размер памяти каждой страницы фиксирован, для компьютеров с архитектурой X86 - 4 КБ (но бывает и больше)

Каждая плита может иметь от 1 страницы до максимум 32 (128/4) страниц (*инфра неточная*)

Когда программа создает кэш, она выделяет ряд объектов в него. Их количество зависит от размера связанных slab'ов. Slab может находиться в одном из следующих состояний:

1. **пустой** — все объекты в slab'e помечены как свободные
2. **частично занятый** — slab содержит как используемые, так и пустые объекты
3. **заполненный** — все объекты в slab'e помечены как используемые

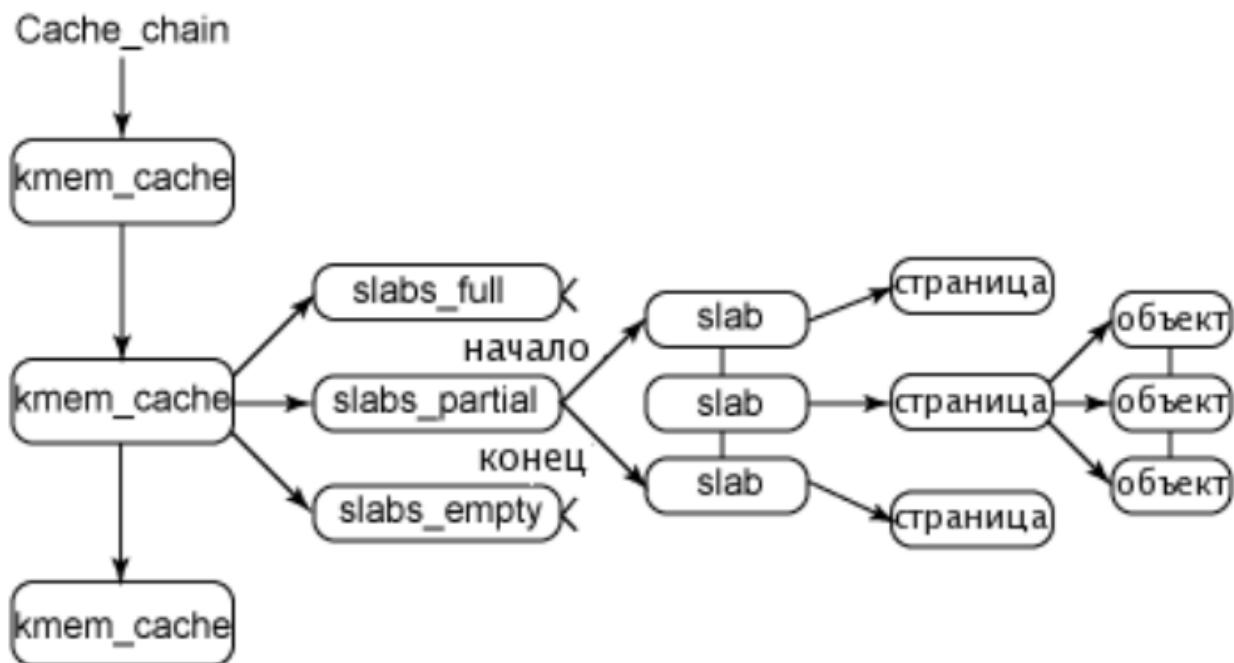
Шаги по версии презентации:

- ядро ищет в кэше объектов объекты типа А и сначала пытается найти частично заполненный кусок и выделить новый объект в нем
- если такого куска нет, то оно просматривает список пустых кусков
- наконец (при необходимости) оно выделит новый кусок, поместит в него новый объект А и свяжет этот кусок с кэшем объектов типа А

Шаги по версии википедии

- Изначально система помечает каждый slab как «пустой».
- Когда процесс обращается за новым объектом ядра, система делает попытку найти свободное место для этого объекта в частично занятом slab'e в кэше для этого типа объектов.
 - Если такого места не находится, система выделяет новый slab из смежных физических страниц и передает их в кэш.
- Новый объект размещается в этом slab'e, а это местоположение помечается как «частично занятое» (или заполнено до конца, зависит от обстоятельств, надеюсь понятно).

Схема SLAB



kmem_cache - интерфейс быстрого буфера памяти (кэш), предоставляемый ядром Linux (должны быть одинакового размера, почему - не могу сказать)

slabs_full/partial/empty - те самые состояния

slab - это определенное пространство памяти kmem_cache.

Страница имеет небольшое пространство, и плита будет вмещать несколько объектов, чтобы максимально использовать пространство.

Пример разбиения на чанки



The principle of buffering records in Slab

The following describes how memcached selects slab for the data sent by the client and caches it in the chunk.

Memcached selects the slab that best fits the data size based on the size of the data received (Figure 2). Memcached holds a list of free chunks in slab, selects chunks based on the list, and then caches the data in it.

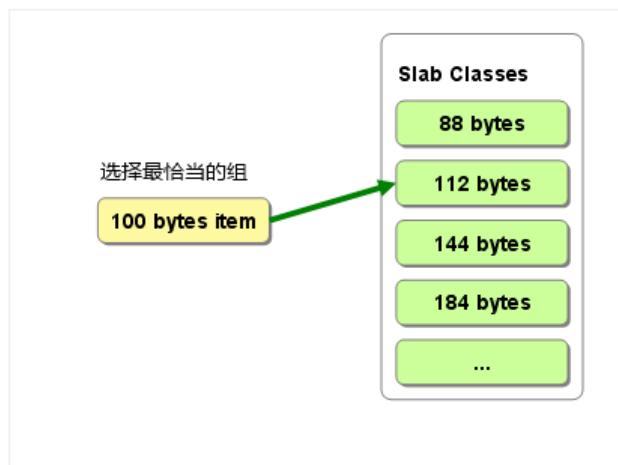


Figure 2 Method of selecting a group to store records

In fact, Slab Allocator is also beneficial and disadvantageous. Here are some of its shortcomings.

способ выбора класса для размещения

Плюсы

- память выделяется ровно в том объеме, в котором требуется, внутренняя фрагментация отсутствует
- Распределение происходит быстро, поскольку система создает объекты заранее и легко выделяет их из slab'a.

Минусы

- Проблема в том, что выделенная память не может быть эффективно утилизирована, потому что она выделяет определенный объем памяти.

Альтернативы

- **SLOB** - для распределения малой памяти, использование в небольших системах, в т.ч. встроенные (Embedded). Использует first-fit алгоритм
- **SLUB** - (усовершенствованный SLAB) механизм управлению памятью, предназначенный для эффективного выделения памяти для объектов ядра, который демонстрирует желательное свойство устранения фрагментации, вызванной выделением и освобождением памяти. [Хранит метаданные](#)

Выделение памяти блоками фиксированного размера (explicit free list)

Вы должны понимать, в чем идея Implicit Free List и метода двоичных маркеров. По хорошему просмотреть презентацию с последней практики с 1 по 36 слайд

[Dynamic Memory Allocation.pdf](#)

Описание

Улучшаем идею Implicit Free List: строим двусвязный список из свободных блоков.

□ **Двусвязный список** - это структура данных, которая состоит из узлов, которые хранят полезные данные, указатели на предыдущий узел и следующий узел.

- Доп. информация про двусвязный список <https://ru.wikipedia.org/wiki/XOR-D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA>

===== Очень хорошее видео, где всё объяснено за 7 минут, правда на английском

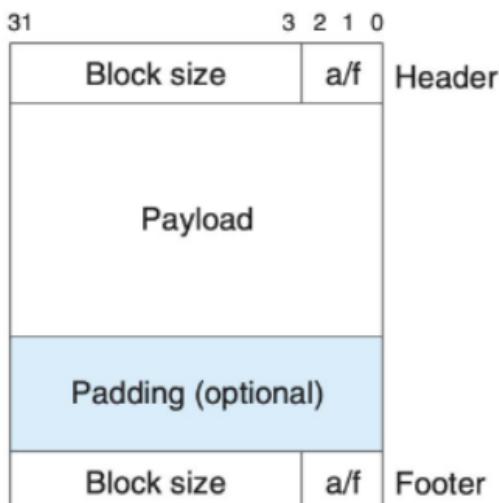
Я брал много картинок оттуда. По сути, можете посмотреть и дальше просто пробежаться по комментариям

Это было и в Implicit Free List:

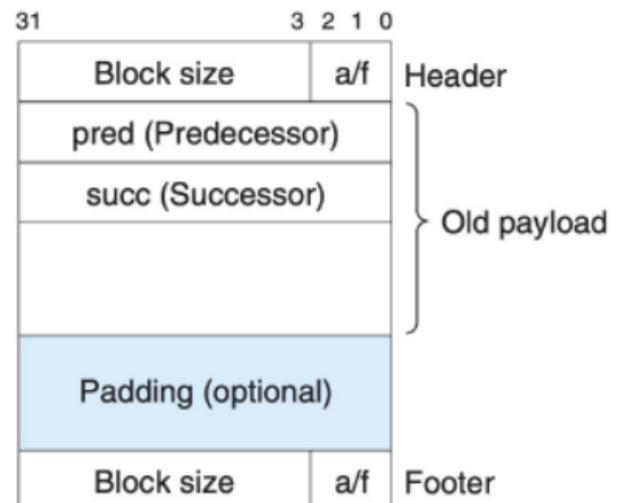
Payload - полезная нагрузка (данные, которые мы храним) **Padding** - просто **набивка**, тут нет полезных данных

Это мы добавили для свободных блоков:

- Pred (или prev от previous) - указатель на *предыдущий свободный* блок
Succ (или next) - указатель на *следующий свободный* блок



(a) Allocated block



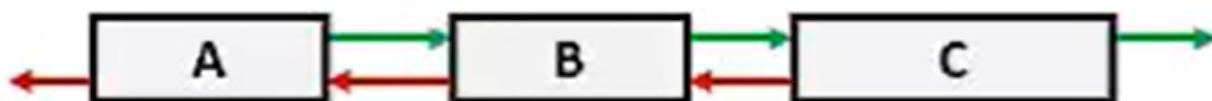
(b) Free block

source: *Computer Systems: A Programmer's Perspective*

Теперь у нас есть указатели, получаем двусвязный список!

Как мы себе это представляем

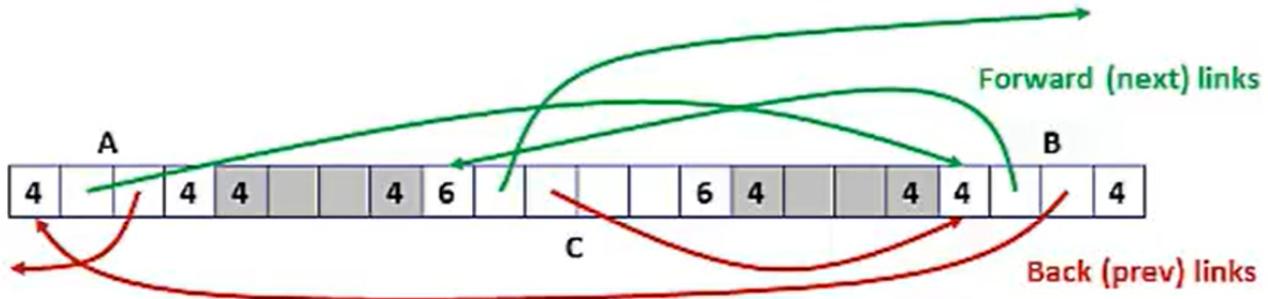
(A, B, C) - свободные блоки



source: <https://youtu.be/rhLk2lf6QXA>

Как это реально хранится в памяти

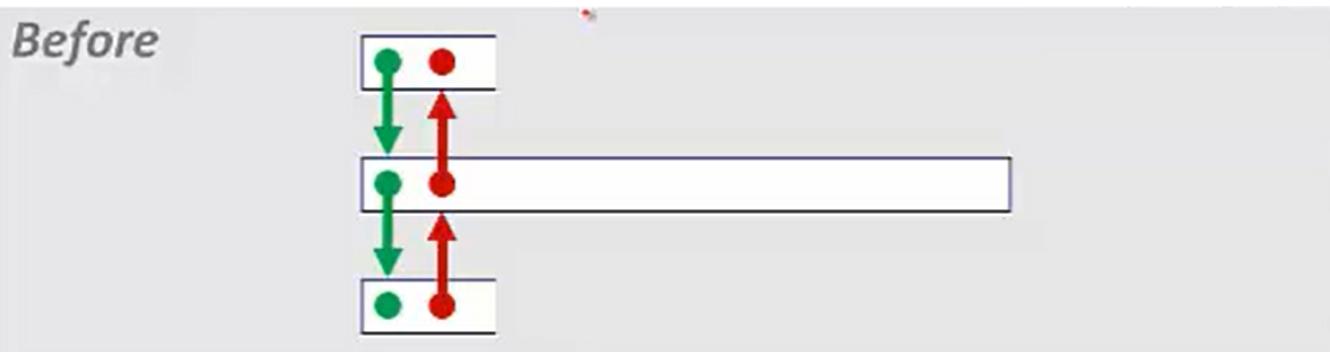
Смысл картинки ниже в том, что свободные блоки с картинки выше в памяти хранятся далеко не последовательно, но благодаря указателям мы можем перемещаться к соседним блокам



source: <https://youtu.be/rhLk2Jf6QXA>

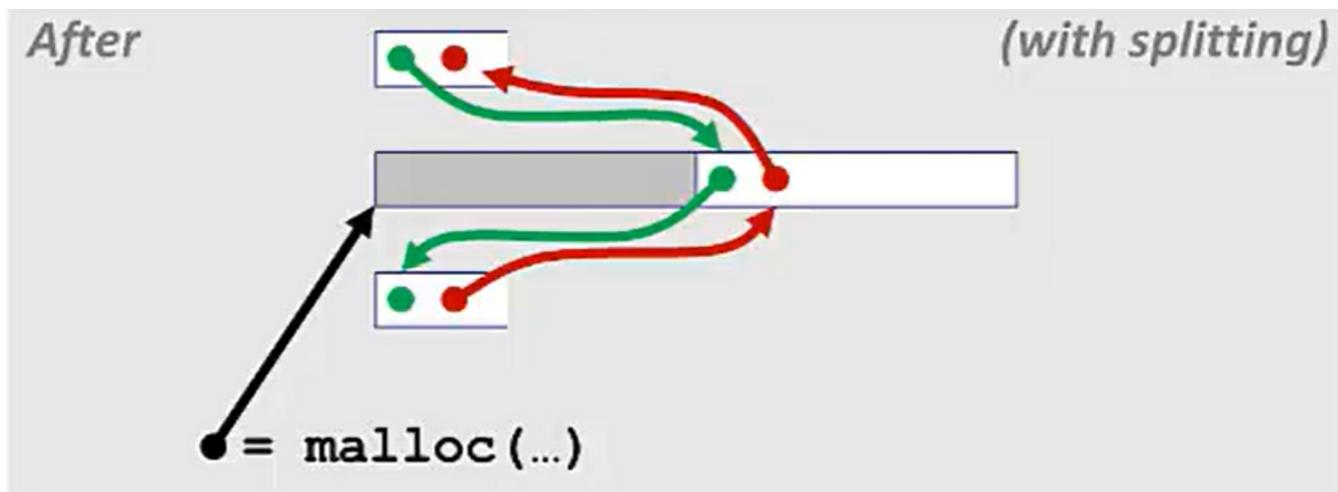
Аллокация блока

- Также, как в Implicit Free List



source: <https://youtu.be/rhLk2Jf6QXA>

Тут мы в большом блоке выделили часть памяти (указатель на эту память возвращает malloc). Теперь мы должны пересчитать этот блок, т к он изменился. Пересчитываем указатели как на картинке.



source: <https://youtu.be/rhLk2Jf6QXA>

Освобождение блока

тут чуть-чуть сложнее

Есть разные политики вставки свободного в список

- Last In First Out (LIFO) - всегда вставляем новый блок в начало
 - free работает за константу
 - Получаем внешнюю фрагментацию
- Address order - блоки в списке упорядочены по адресам.

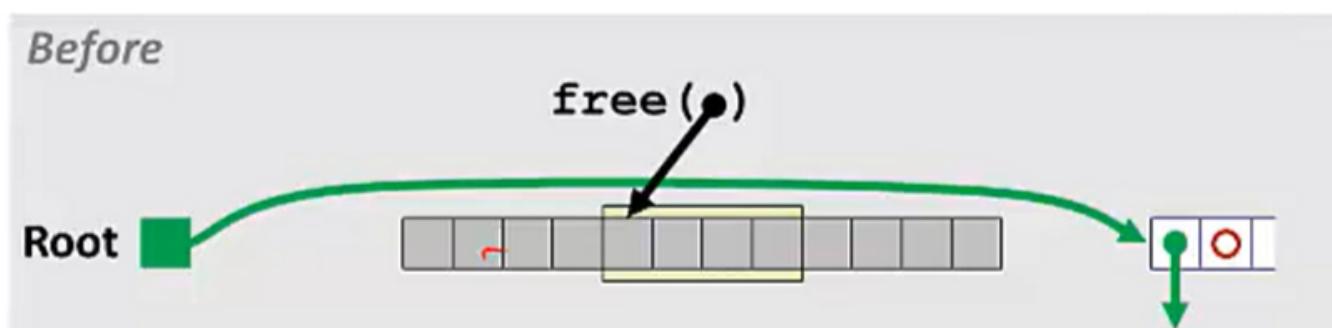
адрес предыдущего < адрес текущего < адрес следующего free работает за линию - проходим по списку, чтобы найти место, куда вставить новый свободный блок trade-off: мы лучше утилизируем память: проход first fit приближается к best fit! фрагментация ниже, чем в LIFO

- Метод граничных маркеры все еще нужен для объединения блоков Опять отсылка на вопрос 38!

Примеры освобождения с подходом LIFO

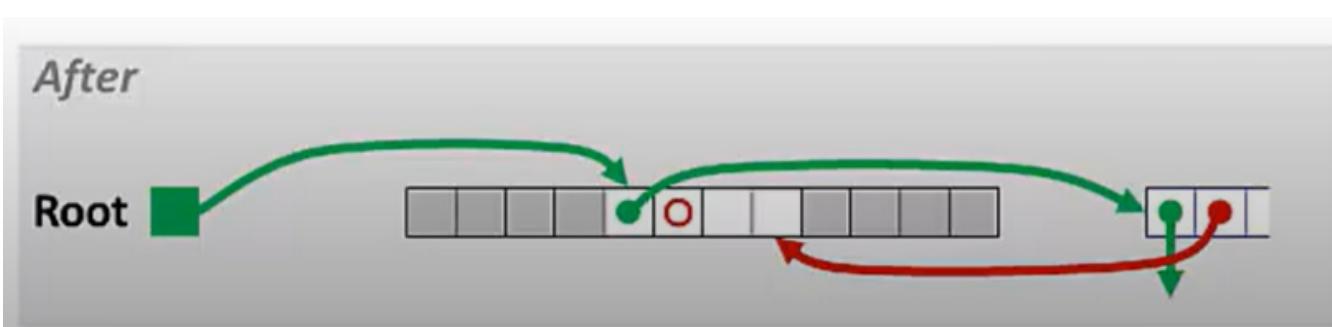
Случай 1

Root - указатель на начало списка. Вставить блок в начало - просто изменить значение root.



Как Вставляем блок в начало:

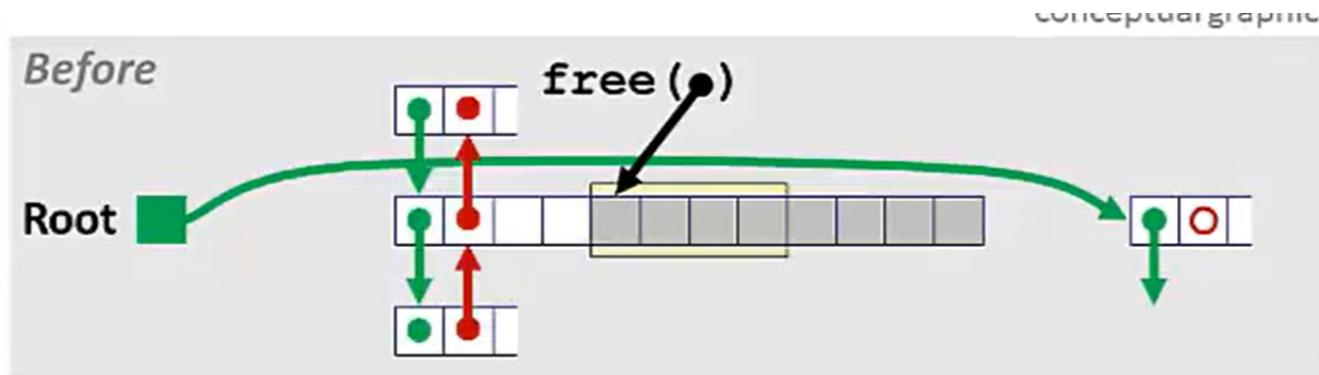
- Root = адрес освобожденного блока
- У того блока, который был первым до нашей вставки (а сейчас стал вторым) обновляем указатель на предыдущий блок



Случай 2-4

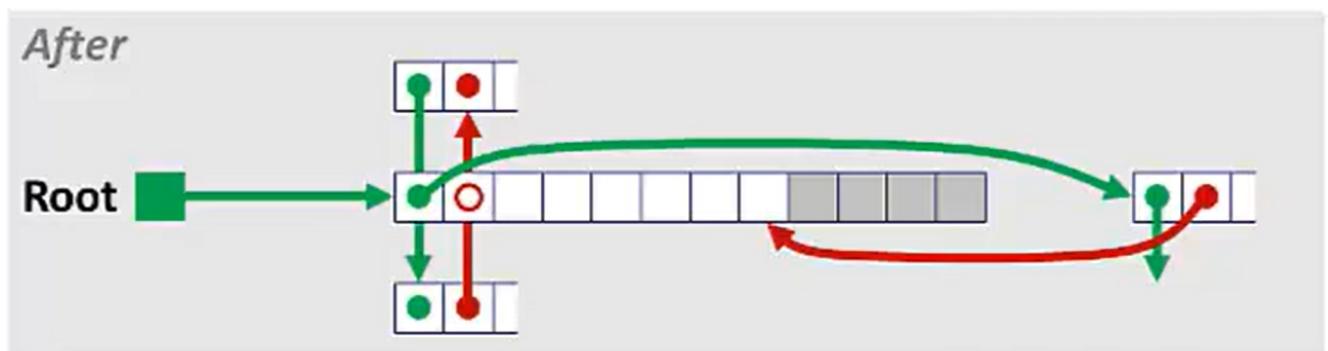
На видео эти ситуации выделены в отдельные случаи, но на самом деле это обычное освобождение блока, как в случае 1 с дальнейшим объединением свободных блоков. Делаем объединение как в методе граничных маркеров, только теперь ещё пересчитываем указатели, которые изменились.

1. Освобожденный блок объединяется с предыдущим

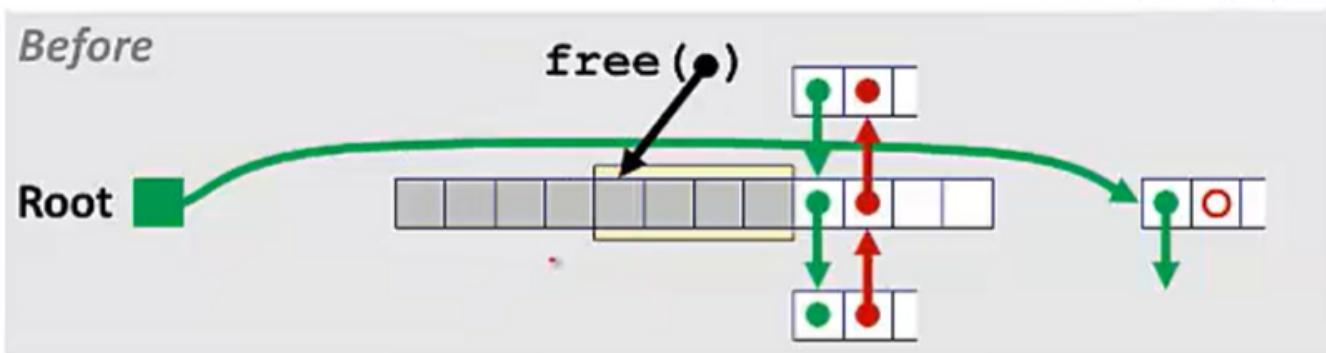


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

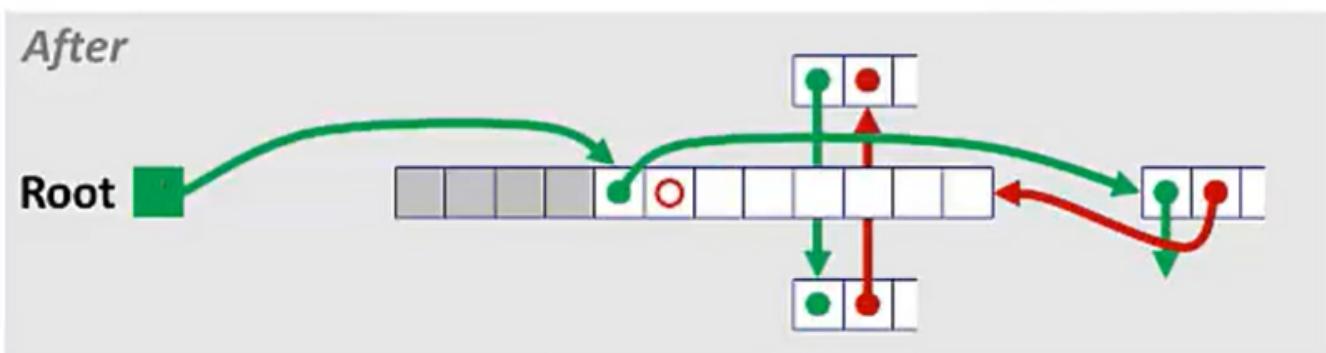
source: <https://youtu.be/rhLk2lf6QXA>



1. Освобожденный блок объединяется со следующим

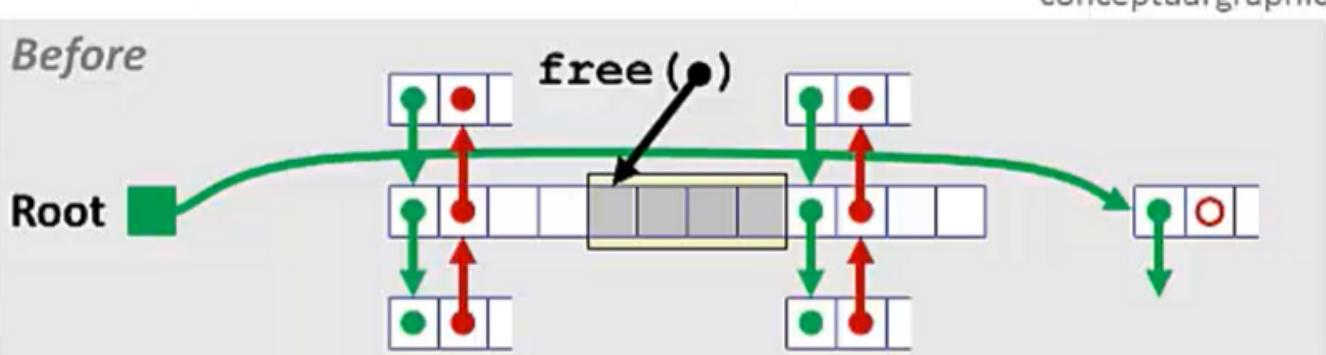


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

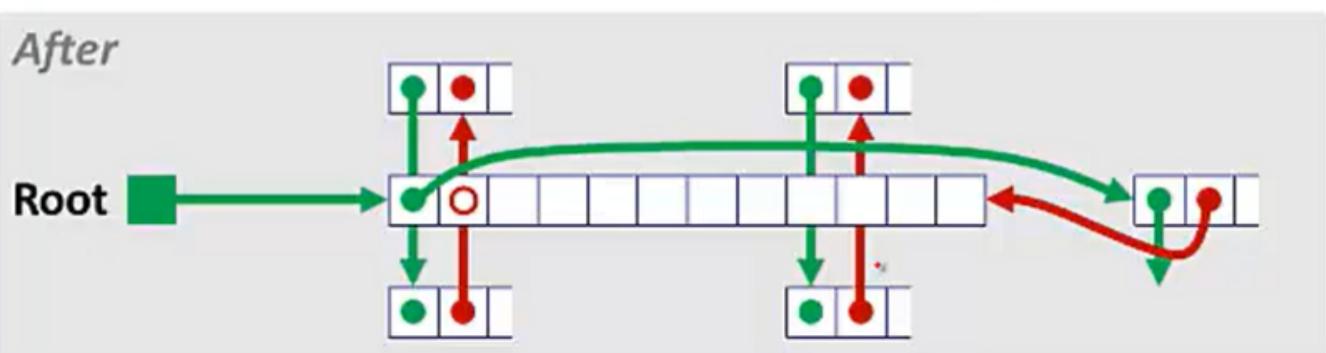


source: <https://youtu.be/rhLk2lf6QXA>

1. Освобожденный блок объединяется и с предыдущим, и со следующим



- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list



source: <https://youtu.be/rhLk2lf6QXA>

Итоги

- ещё раз советую посмотреть [видео](#)
- По сравнению с Implicit Free List:
 - Улучшили время аллокации: линейное время от количества свободных блоков, а не от количества всех блоков
 - Это намного быстрее, когда большая часть памяти заполнена
 - Нужно больше места на свободный блок, ведь мы храним указатели
 - Это может увеличивать внутреннюю фрагментацию >:(

Симметричное шифрование: принцип организации, преимущества и недостатки. Сети Фейстеля. Алгоритмы шифрования DES и 3DES.

Симметричное шифрование — это способ шифрования данных, при котором один и тот же ключ используется и для зашифровывания, и для расшифровывания информации.

Принцип работы симметричных алгоритмов:

Например, если алгоритм предполагает замену букв числами, то и у отправителя сообщения, и у его получателя должна быть одна и та же таблица соответствия букв и чисел: первый с ее помощью шифрует сообщения, а второй — расшифровывает.

Схема симметричного шифрования наглядно (объяснение с практики):

1. Боб посыпает Алисе ключ шифрования key и функции шифрования Encrypt и расшифрования Decrypt
2. Алиса: $\text{Encrypt}(\text{message}, \text{key}) = \text{code}$ (Алиса зашифровывает сообщение, которое хочет послать, ключом, который ей прислал Боб, в итоге она получает некое зашифрованное сообщение-код)
3. Алиса посыпает зашифрованное сообщение code Бобу
4. Боб: $\text{Decrypt}(\text{code}, \text{key}) = \text{message}$ (Боб расшифровывает сообщение по тому, что знает: с помощью какой функции Encrypt зашифровала Алиса, у него есть функция Decrypt и у него есть ключ)

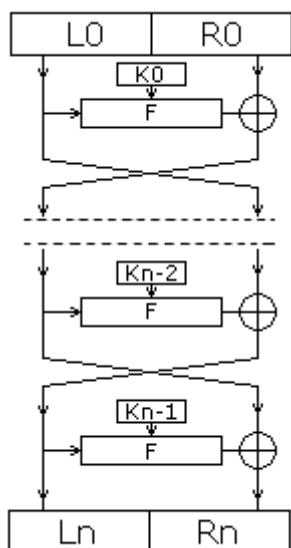
Преимущества симметричных алгоритмов:

- простота реализации
- большая изученность

- меньшая длина ключа для сопоставимости стойкости

Главный недостаток: Считается ненадежным, передаваемый по каналу ключ и алгоритм шифрования может быть перехвачен

Сети Фейстеля — один из методов построения блочных шифров. Сеть состоит из ячеек, называемых **ячейками Фейстеля**. На вход каждой ячейке поступают данные и ключ. На выходе каждой ячейки получают изменённые данные и изменённый ключ. Все ячейки однотипны, и говорят, что сеть представляет собой определённую многократно повторяющуюся (итерированную) структуру.



source: https://en.wikipedia.org/wiki/File:Feistel_encryption.png

Ключ выбирается в зависимости от алгоритма шифрования/расшифрования и меняется при переходе от одной ячейки к другой. При шифровании и расшифровании выполняются одни и те же операции; отличается только порядок ключей. Ввиду простоты операций сеть Фейстеля легко реализовать как программно, так и аппаратно.

<https://habr.com/ru/post/140404/>

DES (Data Encryption Standard) — алгоритм для симметричного шифрования, разработанный фирмой IBM. Размер блока для DES равен 64 битам. В основе алгоритма лежит **сеть Фейстеля** с 16 циклами (раундами) и ключом, имеющим длину 56 бит. Алгоритм использует комбинацию нелинейных (S-блоки) и линейных (перестановки E, IP, IP-1) преобразований.

Алгоритм DES:

- данные переводятся в двоичный код
- разбиваются на блоки по 64 бита
- каждый блок по 16 раундов
- ключ шифрования 56 бит
- зашифрованный блок тоже 64 бита

То есть: взяли сообщение, разбили его на блоки по 64 бита, каждые 64 бита зашифровали,

это все сделали в течение 16 раундов, в которых мы проводили какие-то махинации с этими штуками и в итоге получили то же самое сообщение такой же длины, только зашифрованное)

Схема DES



source: https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/1011/objects/4/files/4_1.jpg

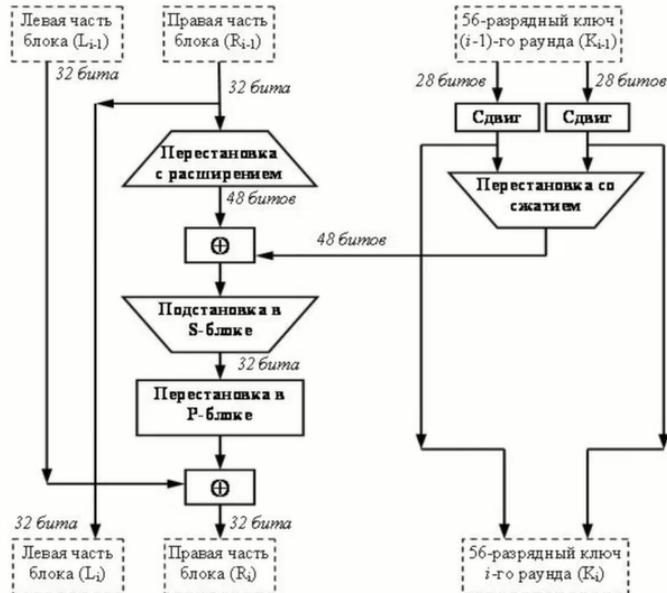
<https://www.youtube.com/watch?v=cR5p9zxK5wY> можно посмотреть еще тут если у вас есть время (а его скорее всего у вас нет)

Формирование частичных ключей (этого нет в билете, думаю можно расписать оч поверхно и не пугаться картинок)

Хотим информацию в 56-битном ключе разбить на много частей, чтобы было сложнее зашифровать и расшифровать, таким образом мы увеличиваем и сложность расшифровки и перехвата сообщения

- 56 битный ключ делится на две половинки
- половинки сдвигаются на 1 или 2 бита(в зависимости от номера раунда)
- после этого выбираются 48 из 56 получившихся бит в определенной последовательности

Схема раунда



source: https://intuit.ru/EDI/20_07_20_2/1595197216-9970/tutorial/1011/objects/4/files/4_2.jpg

Расшифровка DES - выполнение аналогичной последовательности действий

Отличие: частичные ключи используются в обратном порядке

Основной недостаток алгоритма DES:

Длина ключа всего 56 бит, для взлома ключа нужно перебрать (2^{56} вариантов)

Инфа для общего развития: 2^{112} считается хорошей степенью защищенности

У этого алгоритма нет прямых использований, но есть усовершенствования :

Алгоритм **Triple DES (3DES)**.

Вопрос на дом: почему не сработал 2DES ?

В 3DES шифрование/расшифровка выполняются путём троекратного выполнения алгоритма DES. (Был еще Double DES, но он тоже под атаку легко подвергался)

Алгоритм: сообщение кодируется с помощью первого ключа, потом декодируется с помощью второго и снова кодируется с помощью третьего ключа), т.е., мы применяем функции Encrypt и Decrypt 3 раза, которые делают какие-то битовые операции, т.е нам по сути не важно каким ключом мы расшифровываем, мы просто применяем некоторую функцию, которая что-то делает с данными.

По сути это просто усложнение алгоритма DES и усложнение итогового результата зашифрованного сообщения.

А если все три ключа равны получится обычный DES

3DES все еще используется и поддерживается в современном мире (например в роутере у челя с практики, который нам рассказывал про шифрование), но уже переходят на алгоритм AES который быстрее и кручее

Асимметричное шифрование: принципы организации. Алгоритм шифрования RSA. Другие алгоритмы асимметричного шифрования.

Асимметричное шифрование

Асимметричное шифрование основано на парах ключей. Один из них открытый ключ, который доступен всем. С помощью этого ключа кто угодно может зашифровать сообщение. Но для расшифровки берут второй - закрытый ключ. Он должен быть секретным.

Это не могут быть два случайных ключа. Открытый и закрытый ключ всегда связаны между собой алгоритмом, который их выдаёт.

Схема

```
code := Encrypt(mes, pub_key)
```

```
mes := Decrypt(code, priv_key)
```

mes - текст сообщения code - зашифрованное сообщение pub_key - открытый ключ priv_key - закрытый ключ

1. Боб создает пару ключей открытый и закрытый
2. Передает Алисе открытый ключ
3. Алиса зашифровывает свое сообщение с помощью этого ключа
`Encrypt(mes, pub_key)`
4. Отправляет Бобу зашифрованное сообщение
5. Боб расшифровывает его с помощью закрытого ключа `Decrypt(code, priv_key)`

Плюс асимметричного шифрования в том что если кто-то прослушивал канал связи, то он обладает только code (зашифрованным сообщением) и pub_key (открытым ключом). Чего недостаточно для получения mes (текста сообщения).

Аналогия на пальцах

1. У Боба есть ключ (приватный ключ) и коробка с замком (публичный ключ)
2. Боб передает Алисе коробку
3. Алиса кладет в нее сообщение и закрывает
4. Алиса передает закрытую коробку бобу
5. Боб открывает ее своим ключом

RSA

RSA (от фамилий Rivest, Shamir и Adleman) - первый известный алгоритм асимметричного шифрования. Он основывается на сложности разложения на множители произведения больших простых чисел.

Генерация ключей

1. Выбираем два различных больших (>2048 бит) простых числа p и q .
2. $n = p * q$ n - модуль
3. вычисляем функцию Эйлера от n : $f(n) = (p-1) * (q-1)$
4. Выбираем e такое что: $1 < e < f(n)$ и $\text{gcd}(e, f(n))=1$
5. Вычисляем d такое что: $(d * e) \% f(n) = 1$

Пара (e, n) открытый ключ, пара (d, n) закрытый.

Decrypt/Encrypt

Пусть сообщение выражено неотрицательным числом $\text{mes} < n$ (если что сообщение можно разбить на несколько таких блоков).

Encrypt: $\text{code} := (\text{mes} ^* e) \% n$

Decrypt: $\text{mes} := (\text{code} ^* d) \% n$

Почему работает

Тут много математики, так что надеюсь это не нужно, но вот ссылочка.

$c = \text{code}$

$m = \text{mes}$

https://ru.wikipedia.org/wiki/RSA#Корректность_схемы_RSA

Другие алгоритмы

- Криптосистема Рабина - в основе сложность поиска корня из составного числа.
- Схема Эль-Гамаля - в основе сложность вычисления дискретных логарифмов в конечном поле.
- Криптосистема Уильямса - математически более сложный RSA. Однако для которого было доказана необходимость разложить n на $p \cdot q$, для этого RSA доказано не было (хотя хз насколько актуальная инфа).
- Ранцевая криптосистема Меркла — Хеллмана - одна из первых асимметричных криптосистем, но оказалась криптографически нестойкой.

Вроде используются только для [подписи](#) и вообще странные, но на всякий случай:

- DSA - Digital Signature Algorithm = алгоритм цифровой подписи
- ECDSA (Elliptic Curve Digital Signature Algorithm) - на эллиптических прямых

Протокол Диффи-Хеллмана(Меркла). Принципиальное устройство цифровой подписи.

Диффи-Хеллман

Протокол Диффи-Хеллмана - криптографический протокол, позволяющий двум и более сторонам получить общий секретный ключ, используя незащищенный от прослушивания канал связи. Полученный ключ используется для шифрования дальнейшего обмена с помощью алгоритмов симметричного шифрования.

Схема

- Алиса выбирает два числа g и p
 - p - большое простое число (> 2048 бит)
 - g - [первообразный корень по модулю \$p\$](#) (для криптостойкости)
- Алиса отправляет g и p Бобу
- Алиса и Боб выбирают секретные числа (~ 512 бит) (a и b соответственно)
- Алиса и Боб вычисляют $X = (g^{**x}) \% p$, где x – a или b .
- Алиса и Боб обмениваются полученными данными (числа A и B соответственно)
- Алиса и Боб используют полученное число и секретное число для вычисления общего ключа $K = (Ab) \% p = (Ba) \% p = (g^{**(a * b)}) \% p$ $K = (A^b) \text{ mod } p = (B^a) \text{ mod } p = (g^{ab}) \text{ mod } p$
- Общий ключ K дальше используется для открытого шифрования между ними

Подробнее: https://ru.wikipedia.org/wiki/Протокол_Диффи_—_Хеллмана#Описание_алгоритма
или <https://www.securitylab.ru/analytics/478912.php>

В ходе такого обмена ключами, злоумышленник прослушивающий канал не может вычислить финальный общий ключ **K**.

Цифровая подпись

Это механизм подтверждения:

- отсутствия искажения информации в электронном документе с момента формирования подписи
- авторства электронного документа

!! документ - это что угодно, любой файл, сообщение и любой набор байт.

В отличие от асимметричных алгоритмов шифрования, в схемах цифровой подписи закрытый ключ это ключ для зашифровывания сообщения, а открытый наоборот для расшифровывания.

Для того, чтобы использование цифровой подписи имело смысл, необходимо выполнение двух условий:

- Проверка подписи должна производиться только открытым ключом, соответствующим закрытому ключу.
- Без закрытого ключа - вычислительно сложно (невозможно) создать правильную цифровую подпись (даже имея открытый).

Как это работает

- Боб имеет два ключа закрытый и открытый
- Боб рассказывает всем что у него есть открытый ключ **k**
- Дальше когда Боб хочет подписать какой-то документ, он берет зашифровывает его закрытым ключом (или производную от документа, к примеру хеш), и сохраняет это как подпись документа.
 - Далее, документ считается подписанным. И чтобы это проверить, Алиса берет открытый ключ **k** и расшифровывает эту подпись. Если расшифрованный документ совпадает с отправленным, значит:
 - он был подписан Бобом (так как предполагается, что закрытый ключ к открытому ключу **k** есть только у него)
 - никто не поменял содержимое документа

PKI

В основе PKI лежит использование криптографической системы с асимметричным шифрованием и несколько основных принципов:

Инфраструктура открытых ключей (PKI — Public Key Infrastructure) - набор средств (технических, материальных, людских и т. д.), распределённых служб и компонентов, в совокупности используемых для поддержки асимметричного шифрования.

- закрытый ключ (private key) известен только его владельцу;
- **удостоверяющий центр** (CA - Certificate Authority) создает электронный документ - сертификат открытого ключа, таким образом удостоверяя факт того, что закрытый ключ известен только владельцу этого сертификата, открытый ключ свободно передается;
- никто не доверяет друг другу, но все доверяют удостоверяющему центру;
- удостоверяющий центр подтверждает или опровергает принадлежность открытого ключа заданному лицу, которое владеет соответствующим закрытым ключом.

HTTPS

Разберем примерную работу этого на основе протокола https. Когда мы заходим на сайт с https.

- Браузер пользователя просит предоставить сертификат.
- Сайт на HTTPS отправляет сертификат.
- Браузер проверяет подлинность сертификата у удостоверяющего центра.
- Если все ок, браузер и сайт договариваются о симметричном ключе при помощи асимметричного шифрования (на основе сертификата).
- Браузер и сайт передают зашифрованную информацию

И да, прежде чем подключить https к сайту, нужно у удостоверяющего центра для этого сайта запросить сертификат.

Подробнее с https можно ознакомиться в билете:

-
- **Что-то полезное про TLS и HTTP**

TLS (Transport layer security) - универсальный протокол, который способен защитить любое TCP-соединение

В отличие от SSH, клиент не должен заранее знать открытый ключ сервера. Сервер отправляет свой открытый ключ вместе с цифровым сертификатом. Например, если сервер размещен по адресу `example.com`, он должен отправить открытый ключ, а также цифровой сертификат от центра сертификации (**Certificate Authority, CA**), подтверждающий, что `example.com` использует этот открытый ключ.

После этих шагов зашифрованные HTTP-сообщения могут передаваться между клиентом и сервером. Когда TLS используется с HTTP, мы называем его HTTPS. В то время как порт 80 применяется для HTTP, для HTTPS используется порт 443. Веб-серверы, прослушивающие TCP-порт 443, ожидают, что клиент отправит первое сообщение TLS сразу после установления соединения.

TLS используется в нескольких протоколах, помимо HTTPS, включая SMTP. Внешние библиотеки облегчают программистам использование TCP с TLS. Вместо создания обычного сокета TCP программист создает сокет TLS в согласии с внешней библиотекой. Тогда библиотека может взять на себя всю работу по шифрованию. Программист может управлять сокетом TLS почти так же, как сокетом TCP.